VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

CÓMPUTO PARALELO EN LA SOLUCIÓN NUMÉRICA DE LAS ECUACIONES DE BALANCE EN FLUJO TURBULENTO

T E S I S

QUE PARA OBTENER EL GRADO DE:

DOCTOR EN CIENCIAS
(COMPUTACIÓN)

P R E S E N T A:

LUIS MIGUEL DE LA CRUZ SALAS

DIRECTOR DE TESIS: Dr. Eduardo Ramos Mora

México, D.F. 2005.

Cómputo Paralelo en la Solución Numérica de las Ecuaciones de Balance en Flujo Turbulento

Luis Miguel de la Cruz Salas

7 de junio de 2005

Dedicatoria

A mis tres chicas superpoderosas Ange, Michi y Pollis.

A la abuelis y a chayito.

ALF+DM^2

Agradecimientos

Deseo expresar mi agradecimiento a todas la personas que de una u otra manera contribuyeron en el desarrollo de este trabajo, especialmente a mi asesor el Dr. Eduardo Ramos por todos estos años de colaboración y amistad, en los que he aprendido mucho de sus consejos y observaciones. De igual manera agradezco a la Dra. Geneviéve Lucet por su invaluable apoyo en los momentos críticos, no habría llegado a este punto sin su ayuda.

Agradezco el tiempo de los miembros de mi comité tutoral el Dr. Fabián García Nocetti y el Dr. Vladimir Tchijov, sus consejos me permitieron llevar por buen camino este trabajo. También quiero agradecer a mis sinodales, Dr. Ismael Herrera, Dr. Felipe Lara, Dr. Martín Salinas y Dr. Víctor Germán Sánchez, por haber aceptado ser parte del jurado de mi examen, sus comentarios fueron de gran ayuda.

En la DGSCA he conocido a mucha gente con la que he aprendido a ver la vida de otra manera y muchas de ellas influyeron fuertemente en mi decisión de realizar un posgrado en ciencias de la computación. Quisiera recordar a aquellas personas que me apoyaron en mis incios en el mundo de la computación y la tecnología: Alberto Villarreal, Martha A. Sánchez y Rafael Lacambra.

Al personal del Dept. de Visualización, con quienes actualmente colaboro, quisiera darles las gracias por su amistad. De igual manera a mis antiguos compañeros del Dept. de Realidad Virtual, que aunque ya no estamos en el mismo espacio, seguimos siendo muy buenos amigos.

Finalmente, no quisiera olvidar a todos los cuates de Chiconcuac, con los que he corrido aventuras y parrandas que en cierto momento me permitieron quitarme un poco el estrés, Salud! a todos ellos (pelos, carmelo, ganso, gato, gaby, enrique(†), arturo, angel, erick, pitola, paco, mudo, musi, sami, ishiro, simpe, chabelo, torres, y todos los del equipo Huracán).

Resumen

El principal propósito de este trabajo es el desarrollo de un sistema orientado a objetos eficaz, de fácil uso y extensión, para la solución de problemas de dinámica de fluidos. El problema de convección natural en cavidades rectangulares cerradas es utilizado para introducir diferentes aspectos relacionados con la integración de las ecuaciones de Navier-Stokes para fluidos incompresibles. Estas ecuaciones se escriben de manera general con el objetivo principal de describir su versión discreta de forma paramétrica. Esta forma genérica de las ecuaciones incluye parámetros tanto para flujos en régimen laminar como turbulento. La construcción de sistemas para resolver problemas complejos como los que en este trabajo se tratan, necesita de un proceso de desarrollo de software que permita capturar características importantes, como eficiencia, precisión y exactitud, y sobre todo alto nivel de abstracción. El proceso unificado, el cual es usado ampliamente para desarrollar sistemas complejos en diferentes áreas, no toma en cuenta algunos de los requerimientos antes mencionados y además, se basa en la construcción y análisis de casos de uso. En este trabajo se propone un nuevo proceso de desarrollo de software específico para aplicaciones científicas. En este nuevo proceso se incluye el modelo matemático, el modelo discreto, la generalización y la optimización como etapas de desarrollo, y sustituyen a los casos de uso. El resultado final es un conjunto de clases genéricas adaptables con bajo acoplamiento y alta cohesión, las cuales permiten, resolver problemas de convección natural en cavidades rectangulares, en flujo laminar y turbulento, de una manera simple. El sistema contiene además un conjuto de clases para resolver problemas en arquitecturas multiprocesador. En este trabajo se describen los aspectos básicos de programación en paralelo y se diseña un algoritmo simple para resolver problemas que aprovechan arquitecturas mutiprocesadores mediante una metodología de descomposición de dominio. El algoritmo paralelo reutiliza las clases con las que se resuelven los problemas en modo serial. El sistema se probó con varios ejemplos.

Palabras clave: conveccion natural, turbulencia, orientación a objetos, paralelismo.

Abstract

The main purpose of this work is the development of an efficient, easy to use and maintainable object-oriented system for solving computational fluid dynamics problems. Natural convection in rectangular cavities is used as a problem model to introduce different aspects related with the integration of Navier-Stokes equations for incompressible fluids. These equations are written in general form in order to build a parametric description of corresponding discrete equations. The generic form of these equations includes parameters for laminar and turbulent flows. The construction of systems for solving complex problems as those solved in this work, needs a well posed development process to capture important requirements, such as efficiency, precision and exactitude, and mainly with high level of abstraction. The unified process, mostly used in the development of complex systems in many different areas, does not take into account the above mentioned requirements and is also use case driven. In this work a new software development process applied to scientific computing applications is proposed. In this new process the mathematical model, the discrete model, generalization and optimization are included as steps of the whole process to replace the use cases. The final result is a set of adaptable and generic classes with low coupling and high cohesion to solve, in a very simple manner, natural convection problems defined in closed rectangular cavities under laminar and turbulent regime. The system also contains a set of classes to explode parallel architectures. In this work some basic aspects of parallel programming are described and a simple parallel algorithm is devised using a domain decomposition methodology. Classes for solving problems in a single processor are reused in this parallel algorithm. The library was tested with several examples.

Keywords: natural convection, turbulence, object-oriented, parallelism.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Gordon Moore [52] predijo en 1965 que la densidad de transistores en un chip semiconductor podría duplicarse cada 18 meses, de tal manera que la velocidad del procesador, la memoria, el ancho de banda, la capacidad de almacenamiento, etc., deberían tener un incremento del doble cada 1.5 años. Esta regla, que se conoce como la ley de Moore, se ha cumplido durante cuatro décadas y se vislumbra que será válida almenos durante una década más [35]. Sin embargo, a pesar del continuo incremento en el poder de los microprocesadores existe una limitante física: la separación entre los transistores no puede ser menor que el tamaño finito de las partículas atómicas. Actualmente se ha logrado la integración de transistores a una separación de 90 nanómetros (1 nm $= 10^{-9}$ m) y se prospecta llegar a 10 nm para 2015.

Esta limitante física ha ocasionado que se busquen alternativas para resolver problemas de gran escala que requieren procesadores a muy altas velocidades ($\approx$ 10 GHz), memoria muy amplia ($\approx$ TB) y discos de almacenamiento de gran capacidad ($\approx$ PB). Actualmente el uso del cómputo paralelo para resolver este tipo de problemas, que aparecen en distintas áreas de la ciencia, es en la mayoría de los casos la única opción viable para obtener soluciones en tiempos razonables. Particularmente, en las ciencias físicas, el deseo de aproximar soluciones numéricas a problemas cuya solución analítica no ha sido posible encontrar, ha generado el desarrollo de nuevas arquitecturas de cómputo que incluyen procesadores vectoriales y plataformas múltiprocesador.

La construcción de nuevas y más poderosas arquitecturas de cómputo es, en comparación con el desarrollo de software, la parte "fácil" del proceso de simulación numérica. Esta afirmación se basa en el problema conocido como la crisis del software que es la dificultad de construir sistemas simples de entender, mantener y extender, y que además, provean soluciones realistas y con buena precisión a problemas complejos [7, 10]. Este problema no es particular de las simulaciones numéricas y se da en distintas áreas de la ciencia de la computación, en las que un sistema de software robusto es necesario. Diferentes paradigmas de programación han sido desarrollados con el objetivo específico de eliminar el problema de la crisis del software. No obstante, estos "nuevos" paradigmas no se han aplicado directamente a problemas de simulación numérica, debido principalmente a que se tiene la percepción de que el rendimiento se ve afectado sustancialmente.

En la dinámica de fluidos, por ejemplo, el problema de la turbulencia no ha sido resuelto completamente porque existen distintos enfoques teóricos y numéricos [73]. Adicionalmente, para simular problemas de turbulencia se requieren recursos de cómputo que superan a la mayoría de las arquitecturas de supercómputo existentes en nuestros días, véase por ejemplo [41, 51]. El problema anterior se ha intentado resolver a través del uso de arreglos de computadoras personales (*clusters*) y recientemente del uso de Grids [1]. A pesar del impresionante avance en la capacidad de procesamiento y almacenamiento [77], muchas veces no es posible aprovechar este recurso debido a la falta de un desarrollo complementario de software. Actualmente se siguen utilizando códigos bastante complejos, cuyo mantenimiento es complicado y sólo se utilizan para resolver problemas particulares, el reuso es prácticamente nulo.

1.1. Objetivos y metas

El principal objetivo de este trabajo es construir una herramienta para resolver problemas de convección natural en flujo laminar y turbulento, que sea fácil de usar y modificar, y que permita utilizar arquitecturas de cómputo multiprocesador.

Las metas que nos hemos propuesto son las siguientes:

- Realizar la construcción del sistema mediante una metodología de desarrollo de software ordenada he incremental, que permita tomar en cuenta los principales requerimientos de una modelación computacional.
- Escribir de manera general los modelos matemático y discreto de las ecuaciones de balance en flujo laminar y turbulento, para implantar un conjunto de componentes genéricas que puedan reutilizarse en distintos tipos de problemas.
- Generar una arquitectura del sistema simple, que se asemeje a la estructura de una modelación computacional.
- Resolver algunos problemas de convección natural en cavidades rectangulares para calibrar numéricamente el sistema. Comparar con resultados de *benchmarks* conocidos y si es posible con resultados experimentales.
- Resolver los problemas siguiendo una misma metodología, con cambios mínimos.
- Resolver algunos ejemplos usando arquitecturas multiprocesador.

1.2. Dinámica de fluidos computacional

La Dinámica de Fluidos Computacional o CFD por sus siglas en inglés (*Computational Fluid Dynamics*), es una de las áreas de la computación científica que mayor

[1] Organizaciones virtuales que conjuntan equipos de supercómputo muy sofisticado por medio de redes de alta velocidad, como Internet 2 (I2) [33]

crecimiento ha tenido en los últimos años. Esta nueva rama de la dinámica de fluidos, complementa a la teoría y la experimentación, y provee de una alternativa a bajo costo para simular flujos reales. Al igual que en otras áreas de aplicación, la simulación de flujos involucra un proceso que en algunos textos se conoce como modelación computacional, véase por ejemplo [28]. Este proceso se describe en la figura 5.4. En un principio, se tiene el problema en el mundo real. El fenómeno de interés se puede explicar mediante la aplicación de leyes o principios físicos. En este punto se decide a que nivel de aproximación se desea estudiar el fenómeno. Mediante un modelo matemático se traduce el fenómeno del mundo real en un conjunto de ecuaciones, cuya solución permitirá, en principio, entender el fenómeno. En la mayoría de las ocasiones, no existen técnicas matemáticas para resolver analíticamente las ecuaciones que describen el fenómeno bajo estudio. Sólo en casos particulares muy simples se tiene esa ventaja. Cuando no es posible obtener una solución analítica, generalmente se recurre a métodos numéricos para obtener soluciones aproximadas.

En el caso particular que nos ocupa, la dinámica de un flujo se puede describir mediante un conjunto de ecuaciones que incluyen a la ecuación de balance de masa o ecuación de continuidad, las ecuaciones de balance de cantidad de movimiento o ecuaciones de Navier-Stokes y la ecuación de energía, tal como se describe en el capítulo 3. Estas ecuaciones diferenciales parciales, no lineales y acopladas, se conocen desde hace más de un siglo y es bien conocido que su solución puede presentar estructuras a diferentes escalas. La interacción entre todas las escalas influye fuertemente en el comportamiento de la solución. En el lenguaje de Fourier, este comportamiento se describe como la interacción de diferentes frecuencias. La emergencia de la interacción a diferentes escalas se conoce como la transición a la turbulencia, mientras que el estado de interacción total a diferentes escalas recibe el nombre de turbulencia. La distinción entre los regímenes laminar y turbulento depende de diferentes parámetros físicos. En muchos ejemplos importantes, el número de Reynolds conjunta estos parámetros físicos en un parámetro adimensional cuyo valor indica el estado del flujo [72]. Actualmente se conocen algunas propiedades generales de las soluciones a las ecuaciones de balance [15, 43], pero no se conoce un método analítico general de solución En consecuencia, se han sugerido soluciones numéricas aproximadas, tanto en el espacio real, donde las variables independientes (tiempo y espacio) se discretizan para transformar las ecuaciones diferenciales en ecuaciones por diferencias, como en el espacio de Fourier, donde se considera sólo un número finito de frecuencias para expresar las variables dependientes [78].

Cuando se pretende obtener la solución numérica en estado turbulento, se encuentra la dificultad de que existen estructuras a diferentes escalas que se deben resolver simultáneamente. Por lo tanto, la malla del dominio debe ser muy fina, del orden de 10^{15} volúmenes individuales o un número muy alto de componentes de Fourier. Para definir una variable en una malla de tales dimensiones, se necesitarían cientos de Terabytes en memoria, para simular flujos turbulentos de manera útil. Actualmente no existen computadoras capaces de manejar el número resultante de variables y por lo tanto, la simulación directa o DNS (*Direct Numerical Simulation*) no es muy viable en muchos casos. Se debe entonces recurrir a alguna estrategia para incorporar, de alguna man-

era, el efecto de las escalas pequeñas. Se han propuesto varias alternativas para este propósito y una de ellas es conocida como la Simulación de Vórtices Grandes o LES por sus siglas en inglés (*Large-Eddy Simulation*). El método de la LES se originó en estudios de meteorología [66], donde las escalas de resolución numérica requeridas son de aproximadamente 15 órdenes de magnitud. Debido a su éxito relativo, recientemente se ha aplicado a problemas de ingeniería [23]. En esencia, la LES es un filtro pasa bajas, donde las estructuras hasta una cierta escala (umbral) son resueltas por el método numérico con todo detalle y el resto (escalas submalla) son modeladas, [27, 48, 62]. Existen diferentes estrategias para modelar las escalas submalla. En problemas de convección se requiere que estos modelos submalla parametrizen adecuadamente el flujo de calor, de tal manera que las escalas grandes del flujo puedan ser calculadas de manera precisa, y por lo tanto, producir resultados físicamente realistas a un costo relativamente bajo.

1.3. Proceso de desarrollo de software

Los avances en el desarrollo de software han revolucionado muchas actividades de la vida diaria. Desafortunadamente, la mayoría del software existente en nuestros días presenta fallas en alguna etapa de su uso y no entrega soluciones adecuadas a los problemas para los cuales fue creado. Esto sucede principalmente por que los requerimientos del usuario son mal entendidos o cambian durante el desarrollo del software. Además, el mantenimiento, reuso y modificación de algunas partes del software, son tareas complicadas y tediosas.

En la actualidad existen diferentes estrategias para desarrollar software de una manera ordenada e incremental, que capturan desde el principio y durante el desarrollo de un sistema, los requerimientos deseados por los posibles usuarios. El Proceso Unificado [38], que incorpora las ideas de Booch [7], Jacobson [37] y Rumbaugh [63], es actualmente un estándar de desarrollo de software y se utiliza en el diseño de sistemas grandes. Este proceso provee una guía para la construcción eficiente de sistemas de alta calidad y consiste de cuatro fases principales: Incepción, Elaboración, Construcción y Transición, las cuales se descomponen en proyectos pequeños que se van desarrollando en iteraciones. Estas iteraciones consisten de: 1) Requerimientos, 2) Análisis, 3) Diseño, 4) Implementación y 5) Pruebas, de tal manera que en cada iteración se entrega una versión del software que resuelve parte del problema. Durante estas fases e iteraciones se obtiene una arquitectura, la cual contiene los aspectos dinámicos y estáticos más significativos del sistema. La arquitectura conduce la construcción de diferentes módulos y componentes del sistema, y depende de muchos factores (plataforma de cómputo, sistema operativo, componentes existentes, etc.). La arquitectura finalmente es, una vista global del diseño del sistema que muestra las características más importantes, dejando de lado los detalles de la implementación. El proceso unficado es una buena opción para desarrollar software de manera ordenada. Sin embargo, es importante dedicar una buena cantidad de tiempo en el inicio del desarrollo (incepción), que comienza con los casos de

uso. Los casos de uso se encuentran fácilmente en una aplicación interactiva de tiempo real, que no es el caso de los problemas de cómputo científico.

En la construcción de software para cómputo científico, el problema mayor es la realización de un diseño adecuado para que se haga un uso eficiente de las complejas arquitecturas en donde se vaya a ejecutar. Tradicionalmente, el software numérico se realiza usando lenguajes estructurados como Fortran o C. Sin embargo, la complejidad del software crece rápidamente, de tal manera que si al principio el software no está bien ordenado (alta entropía), su tiempo de vida será corto [37]. La Programacion Orientada a Objetos (POO) y las técnicas de análisis y diseño de software, son unas herramientas que pueden ayudar a construir programas numéricos reutilizables, con los cuales se pueden resolver distintos problemas sin muchas modificaciones y con un tiempo de vida largo. Por otro lado, el uso de la POO puede provocar un bajo rendimiento del código con respecto a los programas tradicionales, por lo que se tiene que poner especial cuidado en este problema. El lenguaje C++, que permite programar con diferentes paradigmas (estructurado, orientado a objetos y genérico), ha sido usado en los ultimos años y se ha visto que su herramienta de plantillas (*templates*), permite construir código eficiente, orientado a objetos y genérico [36, 57, 70]. Lo anterior lo hace un lenguaje ideal para aplicaciones de cómputo científico, siempre y cuando se ponga especial atención en las construcciones que introducen fuentes de bajo rendimiento.

1.4. Principales contribuciones

En esta tesis se desarrolló un sistema basado en clases de C++, para resolver problemas de convección natural en cavidades rectangulares. Durante la construcción se combinanron tres paradigmas de programación: Programación Orientada a Objetos (POO), Programación Genérica (PG) y Programación en Paralelo (PP). Las principales aportaciones de este trabajo son:

- Metodología de desarrollo de software para cómputo científico.
  - A partir del modelo de desarrollo propuesto por Schimmel[65] (el cual esta basado en el propuesto por Larman[45] que a su vez utiliza el modelo unificado), se construye una nueva metodología combinándola con la programación extrema (eXtreme Programming).
  - En la nueva metodología se incluyen las etapas de generalización y optimización del sistema, además de las de construcción del modelo matemático y del modelo discreto, incluidas por Schimmel.
- Sistema para solución de problemas de convección natural en cavidades.
  - Arquitectura simple, basada en un proceso general de modelación computacional.
  - Clases generales polimórficas adaptables.

  - Creación de un "metalenguaje" para fácil acceso a las herramientas.
  - Uso de técnicas basadas en "templates" para la optimización.
  - Paralelismo a través de descomposición de dominio, mediante el diseño de un algoritmo paralelo alternante de Schwarz.
- Solución de problemas interesantes.
  - Convección Natural Turbulenta usando LES.
  - Flujo de mezclado.
  - Convección forzada en paralelo.

1.5. Organización de la tesis

La tesis está organizada de la siguiente manara:

- En el capítulo 2 se describe brevemente cada uno de los paradigmas de programación y las técnicas avanzadas de optimización, que se utilizaron en este trabajo.
- En el capítulo 3 se explica el modelo matemático general de las ecuaciones gobernantes de los problemas que resolveremos. Se pone especial atención en el problema de turbulencia. En este último caso se utiliza el modelo matématico de las ecuaciones a filtradas descrito por Bastiaans *et al.* [2] y el modelo submalla de función de estructura selectiva, propuesto por Métais y Lesieur [48]. En este modelo se trabaja en el espacio físico y se mide la viscosidad turbulenta local en cada punto de la malla. Si la magnitud de esta viscosidad turbulenta es menor que un cierto umbral, entonces se elimina y sólo se utiliza la viscosidad molecular, de tal manera que en esos puntos se resuelven básicamente las ecuaciones de flujo laminar. Se escriben las ecuaciones en forma general para usarse posteriormente en el desarrollo del sistema.
- Las ecuaciones se discretizan utilizando el método de volumen finito y se usan diferentes esquemas numéricos para aproximar los términos convectivos. La descripción de estos métodos se encuentra en el capítulo 4. Se escriben los coeficientes de manera general para ser usados en el desarrollo del sistema.
- El sistema se construye siguiendo un proceso de desarrollo de software que es una extensión del proceso descrito en [65]. Este proceso incluye el modelo matemático y la discretización como actividades de desarrollo, y se sustituyen por los casos de uso que se utilizan mayormente en simulaciones en tiempo real y aplicaciones interactivas. La extensión que agregamos es la inclusión de dos etapas más que son la de generalización y optimización, que son dos de los requerimientos que se desea tenga el sistema a desarrollar. Este proceso se combina con el modelo de la programación extrema [4, 86, 85], con el objetivo de que el desarrollo sea más

ágil. Se construye una arquitectura basada en el proceso general de simulaciones numéricas y se reutiliza después en la descripción del algoritmo paralelo utilizado en este trabajo. Todo lo anterior está descrito en detalle en el capítulo 5.

- Finalmente, en el capítulo 6, se dan ejemplos de uso del sistema y se reportan algunos resultados numéricos interesantes.
- En el capítulo 7 se dan las conclusiones, recomendaciones y trabajo futuro.

Capítulo 2

Paradigmas de programación

El alto rendimiento es un requisito indispensable en aplicaciones de cómputo numérico intensivo. La mayoría de los códigos para este tipo de aplicaciones se construyen usando lenguajes estructurados como **F77** o **C**, pues con estos lenguajes se obtiene alto rendimiento de manera simple. Sin embargo, el nivel de abstracción que se puede alcanzar con este tipo de lenguajes es muy bajo, es decir, no existe relación directa de las construcciones del código con entidades del dominio del problema. Esto último ocasiona que el código sea de difícil lectura (aún para el desarrollador) y frecuentemente es muy complicado, y en algunas ocasiones imposible, darle una buena organización, y por lo tanto el mantenimiento y la extensión de dicho código resultan en procesos largos, tediosos y sobre todo costosos. Lo anterior inhibe la reutilización de partes del código para resolver problemas diferentes, a pesar de que en muchas ocasiones se usan algoritmos similares.

La combinación de los paradigmas de programación orientada a objetos (POO), genérica (PG) y paralela (PP), permite construir software con las características necesarias para reutilizarlo en diferentes situaciones y además con un rendimiento comparable al que se obtiene con los lenguajes estructurados. En las secciones que siguen se describen brevemente los conceptos básicos de cada uno de estos paradigmas y algunas técnicas avanzadas, desarrolladas en el lenguaje C++, que eliminan distintas fuentes de bajo rendimiento.

2.1. Programación orientada a objetos

El paradigma de programación orientada a objetos (POO) tiene como principal objetivo el manejo de la complejidad del software. Los conceptos básicos de abstracción, encapsulación, mensajes, polimorfismo y herencia, permiten desarrollar programas de fácil manejo y control, tanto para el desarrollador como para el usuario final. La descripción y funcionalidad de estos conceptos se da a continuación usando el lenguaje C++ para ejemplificar.

2.1.1. Tipos de Datos Abstractos

Algunos autores describen a la POO como programación con Tipos de Datos Abstractos (TDA) y sus relaciones. Un TDA es una abstracción que modela alguna entidad de la vida real y encapsula los datos y las operaciones que se necesitarían para construir dicho modelo. En la POO un problema de la vida real se resuelve mediante la construcción de TDA's que representan entidades bien definidas dentro del dominio del problema. Un TDA consiste de dos partes:

Tipos de Datos Abstractos (TDAs):

Datos o Atributos: En esta parte se describe la estructura de los datos que serán usados para almacenar información.

Operaciones o Métodos: En esta parte se describen las operaciones que serán válidas para el TDA. Un conjtuno de estas operaciones serán la interfaz del TDA con las que el usuario podrá interacturar.

2.1.2. Clases y objetos

Conceptualmente un objeto es una entidad con identidad y con la que se puede interactuar: a un objeto le podemos enviar mensajes y éste reaccionará de alguna manera determinada a mensajes particulares. La reacción a cada mensaje depende del estado interno del objeto, que puede cambiar como consecuencia de la recepción de un mensaje. El objeto con el cual se interactúa tiene un nombre, esto es, tiene una identidad que lo distingue de los demás. Por lo tanto, un objeto tiene comportamiento, estado interno e identidad. Esta definición se debe a Grady Booch [7].

Un objeto es una entidad relacionada con el dominio del problema que se desea resolver: es la abstracción de una entidad del mundo real. Una abstracción significa, en este contexto, enfocar la atención en lo que es fundamental, eliminando o posponiendo las particularidades de la entidad en cuestión. Un objeto puede representar objetos físicos como átomos, células, gráficas, etc.; combinar las características de un conjunto de ellos; o ser simplemente un objeto conceptual que es útil en el desarrollo del sistema. En la implementación, un objeto contendrá algunas características de objetos del mundo real, y características de construcciones conceptuales o computacionales.

La tarea primordial en el desarrollo de un sistema orientado a objetos es la identificación acertada de los objetos que deberán ser implementadas para resolver un problema particular. La interación de los objetos a través del intercambio de mensajes dará como resultado la solución a nuestro problema. Debido a su importancia, se recomienda que la identificación de los objetos se haga mediante alguna de las técnicas especializadas desarrolladas para este propósito, véase por ejemplo [69].

Cada objeto pertenece a una clase de objetos. La clase define las características de los objetos, en este sentido se dice que un objeto es un ejemplo concreto de una clase. En términos de programación, la clase se relaciona con la definición de un TDA, mientras que el objeto con la declaración de una variable:

```
class TipoNuevo { }; \\ definicion de la clase (TDA)
TipoNuevo objeto1;   \\ declaracion de un objeto de la clase A
```

El estado interno de un objeto es el valor de los datos que encapsula. Estos datos se conocen como atributos y pueden cambiar durante el tiempo de vida del objeto.

Un objeto reacciona cuando recibe un mensaje. El mensaje debe estar definido dentro de su clase, de otra manera dicho mensaje no será válido. Los mensajes son peticiones para que el objeto realice una o varias acciones, las cuales se definen en los métodos. Los métodos, desde el punto de vista de la implementación, no son otra cosa que funciones o subrutinas definidas dentro de la clase. Finalmente se dice que un objeto encapsula sus atributos (datos) y sus métodos (operaciones).

Por ejemplo, en el sistema desarrollado en este trabajo, existe la clase EnergyEquation, que es un TDA para describir de manera abstracta, una entidad del dominio del problema (la ecuación de energía en este caso). De esta manera es posible declarar: [1]

```
EnergyEquation energia(T);
```

donde energia es un objeto que interactuará con otros objetos.

2.1.3. Herencia y polimorfismo

La herencia es una de las principales herramientas de reuso de la POO. Una clase puede derivarse de otra a través de la herencia. La nueva clase se conoce como: clase derivada, clase hija o subclase. La clase de la cual se deriva se conoce como clase base, clase padre o superclase. Las clases derivadas se comportan como un subtipo de la clase base y un objeto de la clase derivada puede ser asignado a una variable del tipo de la clase base a través de apuntadores y referencias [70]. Por ejemplo, en nuestro conjunto de clases, hemos definido la clase Mesh para describir a una malla muy general. Luego, para obtener mallas de un tipo particular, se derivan a partir de Mesh, clases con características especiales como StructuredMesh:

```
class Mesh { };                          // Clase base para mallas
class StructuredMesh : public Mesh { };  // Subclase para mallas estructuradas
```

Las clases derivadas pueden ser usadas a su vez como clases base para derivar otras clases más especializadas, de tal manera que es posible construir una jerarquía de clases tan larga y compleja como se quiera. En C++ es posible también derivar una clase de dos o más clases a la vez, esto se conoce como herencia múltiple. La complejidad de las jerarquías, así como la herencia múltiple provocan algunos problemas que afectan principalmente el desempeño del código [8, 70].

El polimorfismo dinámico se realiza mediante la herencia y las funciones virtuales. Consiste en que un objeto de una clase derivada se puede comportar como objeto de la

[1] Esto es similar a declarar una variable en un lenguaje estructurado, por ejemplo en fortran se declara un entero como: integer i.

clase base. Por ejemplo, un algoritmo para hacer la suma de los elementos de una matriz requiere como argumento las entradas de la matriz. Pero la matriz puede ser densa, dispersa, triangular inferior o superior, etc. Las matrices se almacenan en memoria dependiendo de su tipo, de tal manera que no se desperdicie memoria. La consecuencia es que un algoritmo tan simple como el de la suma de las entradas de una matriz, se complica y no es sencillo idear un algoritmo general y eficiente para todas las matrices. Por esta razón, en las bibliotecas numéricas típicas se construyen diferentes subrutinas para cada tipo de matriz. Por otro lado, en la POO, mediante el polimorfismo dinámico, es posible construir una función general que sirva para todos los tipos de matrices [2]. En este caso, cada tipo de matriz tiene que derivarse de una matriz general y el algoritmo que hace la suma recibe como argumento una referencia a una matriz general. Cuando el programa se ejecuta, se determina el tipo de matriz que se necesita y se utilizan los métodos declarados en la definición de dicha clase, por ejemplo:

```
class Matrix {                                          // Clase base abstracta
public: virtual double operator()(int i, int j) = 0;   // Funcion virtual
};
class DenseMatrix : public Matrix {                     //
  public: double operator()(int i, int j) { };          //
};                                                      //
class SparseMatrix : public Matrix {                    //
  public: double operator()(int i, int j) { };          // Clases derivadas
};                                                      //
class TriangularSupMatrix : public Matrix {             //
  public: double operator()(int i, int j) { };          //
};                                                      //

double suma(Matrix &A) {                                // Funcion suma que recibe
   double sum = 0;                                      // una referencia a una
   for(int i=0; i < A.getRows(); ++i)                   // matriz general
      for(int j=0; j < A.getCols(); ++j)                //
         sum += A(i,j);                                 // <- operator()(int i, int j)
   return sum;                                          //
};                                                      //

SparseMatrix S(N,N);                                    // Ejemplo de uso con
double suma_sparse = suma(S);                           // SparseMatrix

TriangularSupMatrix T(N,N);                             // Ejemplo de uso con
double suma_triangular = suma(T);                       // TriangularSupMatrix
```

En el código anterior, en la función suma() se accede a los elementos de la matriz mediante el operador paréntesis: operator()(int i, int j). Este operador debe conocer el formato de almacenamiento de la matriz, por lo tanto, en cada subclase se define un

[2] Esto es un caso típico de programación genérica.

método operator(). En la clase base este operador se declara como virtual para que el programa determine, en tiempo de ejecución, el operador que se usará dependiendo del tipo de matriz. En los ejemplos de uso, el programa determina usar los operadores definidos en SparseMatrix y en TriangularSupMatrix. De igual manera, se podría elaborar una función general para multiplicar matrices que utilice los operadores (operator()) de cada tipo de matriz. Entonces el tamaño del código se reduce de tal manera que su mantenimiento y extensión son simples.

No obstante la utilidad del polimorfismo dinámico, en aplicaciones numéricas puede causar un desempeño muy inferior comparado con códigos de F77/F90. Esto se debe a que el compilador no es capaz de determinar la clase que se está usando en la llamada a la función, pues esta última recibe como argumento una referencia a una matriz general (Matrix). Dentro de la función suma() se utiliza el operador paréntesis (operator()) para acceder a los elementos de la matriz. La determinación de la clase, y por lo tanto del operador que se debe ejecutar se hace en tiempo de ejecución. Este proceso de determinación conocido como ligado tardío (*late binding*) se hace para cada entrada de la matriz. En el caso anterior, si M es el número de columnas y N es el número de renglones, entonces se deben realizar $M \times N$ ligados tardíos, ocasionando que el código sea ineficiente.

El polimorfismo estático soluciona el problema de rendimiento ocasionado por el polimorfismo dinámico. Este tipo de polimorfismo se puede realizar mediante las plantillas de (*templates*) de C++. Las plantillas conducen directamente a la programación genérica y se describen en la sección siguiente.

2.2. Programación genérica

El objetivo principal de la Programación Genérica (PG) es desarrollar programas o componentes genéricas, es decir que se puedan reutilizar en un amplio rango de aplicaciones, manteniendo la eficiencia de los códigos. Mediante la construcción de componentes genéricas se reduce el número de líneas de código que se tienen que implementar. La programación genérica permite construir programas, que en principio, se pueden adaptar a cualquier tipo de dato abstracto. Los programas genéricos involucran un tipo de polimorfismo no tradicional, en el que códigos especializados para diferentes TDA's se obtienen apartir de la parametrización de un programa genérico, el cual utiliza operaciones comunes (requerimientos) a todos los TDA's para realizar las operaciones del algoritmo que implementa. En contraste con programas normales, los parámetros de un programa genérico son estructuras más ricas en contenido, que pueden ser desde tipos básicos (int, char, float, etc.) hasta TDA's muy complejos. La ventaja de este tipo de programación es que se reduce drásticamente el tamaño del código. Por otro lado, debido a la generalidad de los códigos, es probable que algunas operaciones no estén optimizadas para tipos de datos importantes. En este último caso se puede realizar lo que se conoce como una especialización del programa genérico e implementar un caso excepcional para mejorar la eficiencia [81]. En el lenguaje C++, la herramienta de

plantillas o *templates* permite construir código genérico.

2.2.1. Plantillas (*templates*)

Las plantillas o *templates* de C++ son un mecanismo mediante el cual es posible programar usando TDA's como parámetros. De esta manera, es posible que un TDA sea un parámetro en la definición de clases o funciones. Por ejemplo, las entradas de una matriz pueden ser de tipo int o float, dependiendo del tipo de aplicación en donde se usen. Para no repetir la implementación de la clase matriz se usan templates:

```
template<typename data>                          // <---  Parametro
class Matrix { private: data *entradas; };       // <--- uso del parametro

Matrix<double> A(N,N);    // Entradas de tipo double
Matrix<int> B(N,N)        // Entradas de tipo int
```

En el ejemplo anterior, la declaración Matrix<double> genera un ejemplo concreto de la clase Matrix en donde todas las apariciones del parámetro data son substituidas por double. Este proceso de generación de clases se conoce como *template instantation*, y es el compilador el que genera código automáticamente usando el parámetro double. Para cada parámetro distinto se genera una clase con código especializado por dicho parámetro. Algunos compiladores generan código cada vez que se encuentran con la declaración Matrix<parámetro>, aun cuando el párametro se repita, por esta razón, en muchas ocasiones los programas ejecutables que usan *templates* son bastante grandes.

Para las funciones el uso de los *templates* es similar al de las clases, se recomienda consultar Stroustrup [70] para ver más detalles al respecto.

2.2.2. Algoritmos genéricos, conceptos y modelos

La mayoría de los algoritmos se pueden implementar sin poner atención en las estructuras de datos sobre las que éstos operan. Las operaciones que realizan estos algoritmos son comunes para distintos tipos de datos (por ejemplo recorrer y acceder a los elementos de una estructura de datos). Si las estructuras proveen de una interfaz estándar para estas operaciones, se pueden construir algoritmos generales parametrizados con el tipo de estructura de dato mediante los *templates*.

En la biblioteca estándar de plantillas de C++ (STL), que es el ejemplo más conocido de PG, las estructuras de datos se conocen como contenedores y cumplen con ciertos requisitos para funcionar con algoritmos genéricos a través de iteradores contenidos en la misma STL [1, 40, 54]. Por ejemplo, el algoritmo accumulate(), que suma el valor de las entradas de un contenedor, se puede usar como sigue:

```
double x[10];
vector<double> y[10];
list<complex<double>> z(10);
```

```
a = accumulate(x, x+10, 0.0);
b = accumulate(y.begin(), y.end(), 0.0);
c = accumulate(z.begin(), z.end(), 0.0);
```

En este ejemplo se observa como el algoritmo accumulate() puede ser usado independiente de la estructura de datos que se le pasa como argumento.

El buen funcionamiento de un algoritmo genérico depende de la interfaz de las estructuras de datos, es decir, deben cumplir con un conjunto de requerimientos. Cuando una estructura de datos cumple con todos los requerimientos se dice que es un modelo. Las siguientes definiciones son importantes en programación genérica:

Concepto : es un conjunto de características y requerimientos que se deben cumplir para que el algoritmo genérico funcionen correctamente.

Modelo : es un TDA que cumple con las características y los requerimientos planteados en los conceptos.

De aquí que algunos autores definen a la programación genérica como programar con conceptos, donde un concepto es definido por una famila de abstracciones que están relacionadas por un conjunto de requerimientos común. La parte más complicada de la PG es el desarrollo de conceptos, es decir, la identificación de un conjunto de requerimientos lo suficientemente generales para que un grupo grande de abstracciones los cumplan, pero lo suficientemente restrictivos para que los programas se ejecuten eficientemente con todos los miembros del grupo.

Usando algoritmos genéricos y modelos, se reduce el código: si se tienen M algoritmos y N modelos, la cantidad de código que se debe construir en el paradigma de programación estructurada (F77 y/o C) es $M \times N$; por otro lado, en el caso de la POO y la PG con C++, la cantidad de código es $M + N$.

En principio, en la PG se separan los algoritmos de las estructuras de datos, lo que se contrapone con el encapsulamiento de la POO, en donde operaciones y datos están agrupados. Sin embargo, las estructuras de datos se construyen a partir de definiciones de clases de objetos, las cuales encapsulan sus datos y operaciones particulares. Estas últimas operaciones son la interfaz que necesitan los algoritmos genéricos para funcionar y en casos especiales, suplen a los algoritmos genéricos con operaciones más eficientes.

2.2.3. Clases polimórficas

La característica principal de una clase polimórfica es que su funcionamiento puede ser adaptado mediante uno o más TDA's, que son los parámetros de la clase. Los métodos y funciones definidos dentro de la clase polimórfica son ejecutados utilizando métodos definidos en los parámetros, que se conocen como adaptadores: TDA's en donde se definen las operaciones que utilizará la clase polimórfica. Estas operaciones son los requerimientos y los TDA's que los cumplen son los modelos. En este sentido una clase

polimórfica tiene una función similar a la de un algoritmo genérico: es proporcionar una interfaz general para su uso con varios TDA's y reducir el código. Por ejemplo, en el sistema que se desarrolló en este trabajo, existe la clase polimórfica EnergyEquation<class prec, int Dim, class approx> que recibe tres parámetros: prec: tipo de números que se usarán en el cálculo; Dim : dimensión del problema; approx: adaptador que determina la forma de calcular los coeficientes de la ecuación. Este último parámetro permite utilizar la clase con diferentes esquemas numéricos y nos evita la implementación de todo el código, sólo es necesario construir un adaptador para cada esquema numérico.

2.3. Técnicas avanzadas

La POO y la PG proporcionan herramientas que permiten al programador enfocarse en lo que hace el programa y dejar para después los detalles de como lo hace. Sin embargo, estas herramientas, con las que es posible realizar programas con un alto nivel de abstracción, pueden perjudicar sustancialmente el desempeño del código.

El motivo de lo anterior, es que algunas de las construcciones de la POO dificultan tareas importantes del compilador como: reordenación de operaciones, optimización de ciclos (*unrolling*), reuso de memoria (*aliasing*), entre otras. El resultado es un programa ejecutable no optimizado, cuyo rendimiento es muy bajo comparado con códigos de lenguajes estructurados. Diferentes técnicas se han ideado para que los códigos generados sean eficientes, muchas de ellas se pueden consultar en [8, 81].

Uno de los problemas típicos que ocasionan bajo rendimiento es la generación de estructuras de datos y ciclos temporales por el compilador. Por ejemplo, es posible definir una clase array para manejar arreglos de números de cualquier tipo, y además, definir operaciones sobre los arreglos de tal manera que se pueda escribir lo siguiente (en C++):

Código fuente	Seudo código generado por el compilador
`template<class T_number>`	`double * __t1 = new double[N];`
`class array { ... };`	`for(int i=0; i < N; ++i)`
	`__t1[i] = b[i] + c[i];`
`array<double> a(N), b(N),`	`double * __t2 = new double[N];`
`c(N), d(N);`	`for(int i=0; i < N; ++i)`
`a = b + c + d;`	`__t2[i] = __t1[i] + d[i];`
	`for(int i=0; i < N; ++i)`
	`a[i] = __t2[i];`
	`delete [] __t2;`
	`delete [] __t1;`

El código mostrado en la columna de la izquierda, es bastante obvio: se declaran cuatro arreglos y la suma de tres de ellos se asigna al cuarto. En este caso el usuario de la clase array no se distrae en los detalles de los algoritmos para realizar las operaciones sobre los arreglos. Sin embargo, el código generado por el compilador hace uso de por

lo menos dos arreglos temporales y dos ciclos adicionales, para realizar primero la suma __t1 = b + c; luego la suma __t2 = __t1 + d; y finalmente la asignación a = __t2, véase columna de la derecha. Por supuesto que estos arreglos y ciclos temporales no aparecen en el código de la izquierda, pero son generados de manera automática por la mayoría de los compiladores.

Las construcciones como la clase array, son las más utilizadas para mejorar el nivel de abstracción de los códigos de aplicaciones numéricas, pero como se puede observar el rendimiento puede llegar a ser bastante malo. Este es el principal motivo por el cual no se utiliza la POO en aplicaciones de cómputo científico. Afortunadamente existen técnicas que permiten, a través de los *templates*, evitar la aparición de estructuras de datos temporales manteniendo la claridad del código.

2.3.1. Metaprogramación

En 1994, durante una reunión del comité de estandarización de C++, E. Unruh descubrió que los *templates* podrían ser usados para hacer cálculos en tiempo de compilación [80]. A partir de entonces se demostró que el mecanismo conocido como *template instantiation* podría usarse como un lenguaje recursivo primitivo para realizar cálculos no triviales en tiempo de compilación. Este tipo de cálculo es lo que se conoce como metaprogramación [82].

La metaprogramación consiste en construir código parametrizado de tal manera que cuando el compilador lo recorre, sustituye todos los parámetros definidos generando código que implementa la funcionalidad deseada para cada parámeto. Un metaprograma es una parte del código que genera pedazos del código final y que en la mayoría de los casos mejora el rendimiento global del programa.

Traits

Un construcción importante en la metaprogramación es lo que se conoce como *Traits*. Los *Traits* son clases que mapean una cosa en otra. Por ejemplo, la siguiente función parametrizada no funciona si el parámetro T es int, long, char o complex, pues el resultado de la operación es un número flotante.

```
template<class T>
T average(const Vector<T> &datos) {
  T sum = 0;
  int num_elements = datos.getDim()
  for(int i = 0; i < num_elements; ++i)
    sum += datos(i);
  return sum / num_elements;
}
```

Para corregir la función anterior, es necesario realizar un mapeo que produzca un double cuando T es de alguno de los tipos mencionados arriba. El mapeo se realiza mediante el uso de *Traits* como sigue:

```
template<class T>                                          // Trait general
struct float_trait { typedef T T_float;};                  //

template<>                                                 // Trait especializado
struct float_trait<int> { typedef double T_float;};        // para int

template<>                                                 // Trait especializado
struct float_trait<char> { typedef double T_float;};       // para char.
                                                           //... etc
template<class T>
typename float_trait<T>::T_float average(const Vector<T> &datos)
{
  typename float_trait<T>::T_float sum = 0;
  int num_elements = datos.getDim()
  for(int i = 0; i < num_elements; ++i)
    sum += datos(i);
  return sum / num_elements;
}
```

De esta manera el compilador elije el tipo dependiendo de la definición del *Trait*, véase por ejemplo [81].

En aplicaciones numéricas, a menudo se tienen que procesar arreglos multidimensionales, o realizar operaciones aritméticas entre vectores. Por ejemplo el producto punto de dos vectores:

```
template <typename T>
inline T dot_product (int dim, T* a, T* b) {
    T result = T();
    for (int i=0; i<dim; ++i) {
        result += a[i]*b[i];
    }
    return result;
}
int main() {
    int a[3] = { 1, 2, 3};
    int b[3] = { 5, 6, 7};
    int dot = dot_product(3,a,b)
}
```

El código anterior da resultados correctos, pero puede tomar mucho tiempo cuando los arreglos son muy grandes. El problema es que el compilador no puede desarrollar (*unrolling*) el ciclo de la función dot_product() de manera controlada, de tal manera que se genera un código ineficiente. Es posible resolver este problema construyendo un metaprograma que controle el desarrollo del ciclo de manera óptima. Por ejemplo:

```
// Clase template para el producto punto en general
template <int DIM, typename T>
class DotProduct {
  public:
    static T result (T* a, T* b) {
        return *a * *b  +  DotProduct<DIM-1,T>::result(a+1,b+1);
    }
};

// Especializacion parcial: criterio de terminacion
template <typename T>
class DotProduct<1,T> {
  public:
    static T result (T* a, T* b) {
        return *a * *b;
    }
};

// Funcion para producto punto
template <int DIM, typename T>
inline T dot_product (T* a, T* b)
{
    return DotProduct<DIM,T>::result(a,b);
}

int main() {
    int a[3] = { 1, 2, 3};
    int b[3] = { 5, 6, 7};
    int dot = dot_product(3,a,b)
}
```

El resultado del código anterior es que la línea dot = dot_product(3,a,b) se transforma, mediante un proceso recursivo de *template instantation*, en lo siguiente:

```
→ dot = DotProduct<3,int>::result(a,b);
→ dot = *a * *b + DotProduct<2,int>::result(a,b);
→ dot = *a * *b + *(a+1) * *(b+1) + DotProduct<1,int>::result(a,b);
→ dot = *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2);
```

Se observa que al final se tiene un código más eficiente en donde el ciclo se ha desenrollado totalmente. Nótese que en este ejemplo se conoce la dimensión de los vectores antes de la ejecución. Bibliotecas como Blitz++ [6] y uBLAS [79] utilizan este tipo de metaprogramas para generar rutinas eficientes de álgebra lineal.

2.3.2. Expresiones parametrizadas

Una técnica especial de metaprogramación se conoce como *expression templates* (ET) [83]. Mediante esta técnica es posible eliminar las estructuras de datos y ciclos temporales que aparecen en expresiones aritméticas entre arreglos, transformándolas en ciclos eficientes. El proceso que se sigue en la transformación de expresiones se ilustra en la figura 2.1. El detalle de esta técnica se puede consultar en [81, 83]. En la actualidad, para obtener buenos rendimientos es indispensable incluir ET en los kernels numéricos de sistemas orientados a objetos, como es el caso de Blitz++ [6] y uBLAS [79].

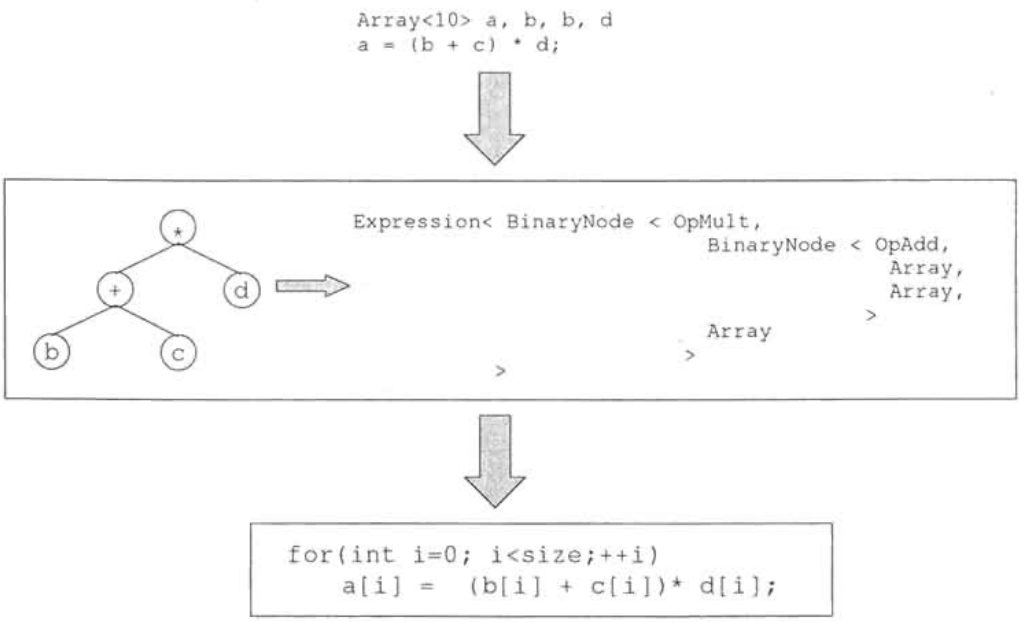


Figura 2.1: Expresiones parametrizadas: un código que expresa fielmente la operación aritmética que implementa, es transformado por un ET y el compilador en un ciclo eficiente.

2.3.3. Herencia parametrizada recursiva

Usando herencia podemos construir algoritmos genéricos que reciban como argumento una referencia a un objeto de la clase base y utilizar las operaciones definidas en las clases derivadas. Esto se puede realizar con las funciones virtuales, sin embargo, las funciones virtuales ocasionan una sobrecarga en la ejecución del programa. Una manera de evitar esto se puede usar la técnica conocida como *CRTP: Curiously Recurring Template Pattern*, [81].

La idea es como sigue: primero se determinan las operaciones generales de un conjunto de objetos y se agrupan en una clase base; luego, se derivan clases particulares de tal manera que las subclases parametrizen las operaciones generales de la clase base.

De esta manera, las operaciones generales serán modificadas de acuerdo con las clases derivadas y dado que se usan *templates*, las operaciones modificadas se generarán en tiempo de compilación evitando cualquier tipo sobrecarga. Con esta técnica, el ejemplo de matrices descrito en la sección 2.1.3, se transforma como sigue:

```
template<class T_clase_derivada>
class Matrix {
public:
  T_clase_derivada& como_clase_derivada() {        // funcion que determina el
    return static_cast<T_clase_derivada&>(*this);  // tipo de la clase derivada
  }

  double operator()(int i, int j) {
    return como_clase_derivada()(i,j);
  }
};
//
// Clases derivadas
//
class DenseMatrix : public Matrix<DenseMatrix> { };
class SparseMatrix : public Matrix<SparseMatrix> { };
class TriangularSupMatrix : public Matrix<TriangularSupMatrix> { };
//
// Uso de las clases
//
template<class T_clase_derivada>            // Esta funcion recibe una
double suma(Matrix<clase_derivada>& A) {    // Matriz de la clase base
   double sum = 0;                          // parametrizada con una clase
   for(int i=0; i < A.getRows(); ++i)       // derivada
      for(int j=0; j < A.getCols(); ++j)
         sum += A(i,j);
   return sum;
};

SparseMatrix S(N,N);                // Ejemplo de uso con
double suma_sparse = suma(S);       // SparseMatrix

TriangularSupMatrix T(N,N);         // Ejemplo de uso con
double suma_triangular = suma(T);   // TriangularSupMatrix
```

En el código anterior, la clase Matrix se parametriza con el tipo de la clase derivada. Las subclases heredan de la clase Matrix y usan como parámetro su mismo tipo. Cada clase derivada debe proveer una función operator() para acceder a sus datos, la cual depende del tipo almacenamiento de cada matriz. La ventaja de esta construcción con respecto a la descrita en la sección 2.1.3, es que aquí se genera el código necesario

en tiempo de compilación debido a la parametrización. Nótese que la forma de uso es exactamente la misma.

2.4. Programación paralela

Las computadoras tradicionales, con un solo procesador, están basadas en el modelo introducido por John von Neumann [9]. Este modelo consiste de una unidad central de procesamiento (CPU) y una memoria. Este tipo de computadoras toma una secuencia simple de instrucciones y opera sobre secuencias simples de datos. Las computadoras de este tipo se conocen como SISD (*Single Instruction Single Data*). La velocidad de una computadora SISD está limitada básicamente por dos factores: velocidad del procesador y capacidad de memoria. La velocidad del procesador se incrementa según la ley de Moore [52], pero existe un límite físico: la separación entre transistores no puede ser menor que la separación entre partículas elementales. La capacidad y velocidad de acceso a la memoria se mejoran mediante diferentes tecnologías (que incluyen *cache, memory interleaving* y *pipelining*) pero también existen limitaciones [42].

Una manera alterna para mejorar la velocidad de ejecución es el uso de múltiples CPUs con su memoria, interconectados de alguna manera. En teoría, la razón de procesamiento crecerá conforme se incremente el número de procesadores. Las computadoras que contienen múltiples procesadores se conocen como computadoras parallelas y existen diferentes tipos que se clasifican de acuerdo a su arquitectura. La clasificación de Flynn [21] es la más conocida y consiste de las siguientes categorías:

SISD (*Single Instruction, Single Data*) : Computadoras tradicionales que procesan instrucciones de manera serial.

SIMD (*Single Instruction, Multiple Data*) : Computadoras con varios procesadores que trabajan concurrentemente y ejecutan las mismas instrucciones sobre conjuntos de datos diferentes.

MISD (*Multiple Instruction, Single Data*) : Computadoras con varios procesadores donde cada uno de ellos puede modificar los datos antes de pasar al siguiente procesador, el cual realiza otro tipo de operaciones sobre los mismos datos.

MIMD (*Multiple Instruction,Multiple Data*): Computadoras en las que cada procesador es capaz de ejecutar conjuntos de instrucciones diferentes independiente de los otros procesadores.

2.4.1. Organización de la memoria

Resolver un problema en un ensamble de procesadores requiere de la interacción entre procesadores. Esta interacción está determinada por los dos tipos básicos de arquitecturas: *memoria compartida* y *memoria distribuida.*

Memoria compartida : Los diferentes procesadores comparten un espacio de memoria global a la cual acceden mediante un canal o bus de datos de alta velocidad. Este espacio de memoria global permite a los procesadores intercambiar o compartir datos. El número de procesadores usado en una arquitectura de este tipo está limitado por el ancho de banda del bus de datos que conecta a los procesadores.

Memoria distribuida : Cada procesador tiene su propia memoria local o privada. Todos los procesadores o nodos tienen acceso rápido a su memoria local y pueden acceder a la memoria de otros nodos a través de una red, la cual debe permitir comunicaciones de alta velocidad. Los datos que se intercambian entre los nodos son enviados como mensajes sobre la red.

2.4.2. Modelos de programación

La programación en una arquitectura multiprocesador se puede hacer básicamente de dos maneras:

- Paralelismo de memoria compartida, por medio de directivas. Este tipo de programación puede ser implementado con HPF (*High Performance Fortran*) [30] o con OpenMP [56], donde un código serial se paraleliza adicionando directivas (las cuales aparecen como comentarios cuando se realiza una ejecución serial) que le dicen al compilador como distribuir los datos y el trabajo entre los procesadores. Los detalles de la distribución de los datos y comunicaciones son realizados por el compilador. Estas implementaciones se realizan usualmente en arquitecturas de memoria compartida debido a que el espacio global de memoria simplifica grandemente la escritura de los compiladores.

- Progamación en envío de mensajes, mediante llamadas a funciones. Este segundo enfoque puede implementarse en ambos tipos de arquitectura de memoria. En este caso el programador es responsable de dividir explícitamente los datos y el trabajo entre los procesadores, así como manejar las comunicaciones entre ellos. Este enfoque es muy flexible y portable a diferentes tipos de máquinas. Se puede implementar con PVM (*Parallel Virtual Machine*) [61] y con MPI (*Message Passing Interface*) [24, 68].

2.4.3. Comunicaciones

El hardware que permite a los procesadores comunicarse es un aspecto crítico en ambos tipos arquitecturas de memoria. Desde un punto de vista abstracto no hace mucha diferencia si se conectan procesadores al bus de memoria o si se conectan computadoras entre sí. El papel que juega la red en un cluster de PCs es comparable al papel del bus de datos en una computadora de memoria compartida.

En general, un programador no necesita conocer la topología de interconección de los procesadores para construir un programa paralelo. El software que maneja la red y el hardware de las computadoras paralelas actuales esconden los detalles de bajo nivel de la red, permitiendo a los códigos enviar trabajos a cualquier procesador de la computadora. Sin embargo, es útil tomar en cuenta dos aspectos de las redes que son relevantes para el diseño de un algoritmo.

1. Razón de transferencia de datos. El modelo estándar de una red involucra dos parámetros:

   - La latencia (*network latency*) $L(seg)$, que es el tiempo necesitado para iniciar la conección entre dos procesadores.
   - El ancho de banda (*bandwidth*) $B(\frac{bytes}{seg})$, que es la razón a la cual los datos son intercambiados después de que se ha iniciado una conección.

   El tiempo t_d para transferir b bytes de datos es:

   $$t_d = L + \frac{b}{B} \tag{2.1}$$

   De lo anterior se observa claramente que es mejor enviar un mensaje largo, en vez de un conjunto de mensajes cortos, aún si la cantidad total de datos transferidos es la misma.

2. Comunicaciones locales y globales. Los diferentes modelos de programación paralela permiten comunicaciones uno a uno, todos a uno y todos a todos. Cada uno de estos tipos de comunicación tiene sus costo. Un ejemplo típico es el producto punto entre dos vectores $\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{N} u_i v_i$. Si cada uno de los P procesadores tiene un subconjunto de las componentes de ambos vectores, estos realizan una suma parcial local sin necesidad de comunicaciones. Pero el resultado final involucra necesariamente comunicaciones globales, que dependendiendo de P y L pueden ser más caras que las sumas parciales realizadas en paralelo.

En el sistema desarrollado en este trabajo son necesarios dos tipos de comunicaciones: cada procesador a sus vecinos (para intercambiar condiciones de frontera) y todos a uno (para calcular errores, residuos y generar resultados finales). En el primer caso, el término "vecinos" se refiere a la posición relativa de cada procesador en la topología virtual usada en la descomposición del dominio, véase sección 5.3.

2.4.4. Descomposición del problema

El primer paso en el diseño de un algoritmo paralelo es descomponer el problema en subproblemas más simples. Luego, cada uno de estos subproblemas se asignan a diferentes procesadores en donde se resuelven simultáneamente. Existen básicamente dos tipos de descomposición: descomposición de dominio y descomposición funcional.

Descomposición de Dominio (DD) : En el modelo de descomposición de dominio o paralelismo de datos, los datos se dividen en porciones de aproximadamente el mismo tamaño que se mapean a diferentes procesadores. Cada procesador trabaja con la porción de datos que le fue asignada. En algunos problemas los procesadores necesitarán comunicarse periódicamente para intercambiar información. Un algoritmo en este modelo consiste de una secuencia de instrucciones elementales aplicadas a los datos. La arquitectura SIMD se acopla a este tipo de paralelismo, donde copias del mismo código se ejecutan sobre diferentes porciones de datos en diferentes procesadores. La estrategia DD se emplea comúnmente en algoritmos para resolver ecuaciones diferenciales parciales con valores a la frontera, donde los procesadores pueden operar independientemente sobre porciones de datos grandes, comunicando sólo una pequeña parte de datos (la frontera) en cada iteración.

Descomposición Funcional (DF) : Frecuentemente, la estrategia de descomposición de dominio resulta **no** ser la más eficiente para un programa en paralelo. Esto sucede cuando las porciones de datos asignadas a los diferentes procesadores requieren de espacios de tiempo muy distintos para ser procesadas. Entonces, el rendimiento del código está limitado por la velocidad del procesador más lento. Los procesadores que terminan primero permanecen inactivos hasta que el procesador más lento finaliza su trabajo. En este caso, la descomposición funcional o paralelismo de tareas es más efectivo que la descomposición de dominio. En el paralelismo de tareas, el problema se descompone en un número grande de tareas pequeñas, las cuales se van asignando a los procesadores conforme estós estén disponibles. A los procesadores que terminan su trabajo más rápido simplemente se les asigna otra tarea. El paralelismo de tareas se implementa en un modelo de cliente-servidor. Las tareas se distribuyen entre un grupo de procesadores esclavos por un procesador maestro, el cual también puede realizar algunas tareas. El modelo de cliente-servidor puede ser implementado virtualmente a cualquier nivel del programa. Por ejemplo, si se desea ejecutar un programa con diferentes entradas, una implementación cliente-servidor puede ejecutar múltiples copias del código serial, con el servidor asignando las diferentes entradas a cada proceso cliente. Conforme cada procesador vaya terminando su tarea se le asignará una nueva con una entrada diferente. Un ejemplo donde este tipo de descomposición es útil es en aplicaciones que se ejecutan en tiempo real, que es el caso de sistemas de visualización interactiva y realidad virtual.

2.4.5. Métricas y otros factores

El principal objetivo de un programa paralelo es resolver un problema en menos tiempo comparado con la versión serial del mismo. En el diseño de un código paralelo se deben considerar algunos factores importantes para obtener el mejor rendimiento posible dentro de las restricciones del problema que se esté resolviendo. Para medir el beneficio del paralelismo existen algunas parámetros estándares que son usados ampliamente y

que se describen a continuación.

Balance de carga

El trabajo total de cálculo que se realizará para resolver el problema, debe ser dividido equitativamente entre el número de procesadores disponibles. Esto se hace fácilemente cuando el mismo conjunto de operaciones es realizado por todos los procesadores, sobre diferentes porciones de datos. En el caso de DD el balance de carga es obtenido automáticamente si las porciones de datos son iguales, aunque en algunos casos se deben considerar las condiciones en las que esto es válido. Por ejemplo si un procesador debe resolver la capa límite de un flujo, éste tardará más que otros procesadores que no lo hacen, aun cuando trabajen sobre porciones de datos del mismo tamaño. En el caso de la DF, el balance de carga es más importante, pues es necesario asignar tareas del mismo peso a cada procesador para evitar que haya procesadores ociosos, mientras que otros trabajan al 100 %.

Métricas de rendimiento

Algunas de las métricas que se utilizan frecuentemente para medir el rendimiento de programas paralelos son las siguiente:

Tiempo de ejecución : El tiempo de ejecución de un programa serial se denota por T_s El tiempo de ejecución en paralelo se mide a partir del momento en que el cálculo en paralelo inicia hasta que el último procesador termina su ejecución, y se denota por T_p. El tiempo total de ejecución de un programa en paralelo debe tomar en cuenta el tiempo de cálculo, el tiempo de inactividad y el tiempo de comunicación.

Aceleración (*Speedup*) : Es una medida que captura el beneficio relativo de resolver un problema en paralelo. Se define como la razón del tiempo que se lleva resolver el problema en un solo procesador entre el tiempo requerido para resolver el mismo problema con n_p procesadores idénticos. Entonces, la aceleración se calcula mediante $S = T_s/T_p$. Una aceleración ideal para n_p procesadores sería $S = n_p$.

Eficiencia : La eficiencia mide la fracción de tiempo que un procesador pasa realizando operaciones del algoritmo paralelo, es decir, sin incluir comunicaciones, tiempos de espera, etc. Se define como $E = S/n_p$. Una eficiencia ideal sería $E = 1$.

Costo : Se define el costo de resolver un problema en paralelo como el producto del tiempo de ejecución en paralelo T_p por el número de procesadores usados n_p. Este número refleja la suma del tiempo que cada procesador gasta resolviendo el problema. Se tiene un costo óptimo si es es proporcional a T_s.

Sobreposición de la comunicación y los cálculos

Existen varias formas de minimizar el tiempo inactivo entre los procesadores, y un ejemplo es sobreponer los cálculos con las comunicaciones. Esto implica ocupar un procesador con una o más tareas nuevas mientras se espera a que termine la comunicación con otros procesadores. El uso cuidadoso de comunicaciones no bloqueables (*non-blocking*) hacen posible lo anterior. Sin embargo, es muy difícil, en la práctica, intercalar comunicaciones y cálculos.

Ley de Amdahl

La ley de Amdahl se escribe como sigue:

$$T_p = T_s \left(\frac{P_p}{n_p} + P_s \right) \tag{2.2}$$

donde P_s es el porcentaje del tiempo gastado ejecutando de forma serial; P_p es el porcentaje del tiempo gastado ejecutando en paralelo y n_p es número de procesadores. Además se cumple que $P_s + P_p = 1$. De la ecuación (2.2) tenemos que una medición de la aceleración es:

$$S = \frac{T_s}{T_p} = \left(\frac{P_p}{n_p} + P_s \right)^{-1} \tag{2.3}$$

La siguiente tabla muestra el comportamiento de la aceleración de acuerdo con la ley de Amdahl.

P_p	S (2)	S (4)	S (8)	S (16)
80 %	1.6	2.5	3.3	4.0
90 %	1.8	3.0	4.7	6.4
95 %	1.9	3.5	5.9	9.0
98 %	1.96	3.7	7.0	12.3

Cuadro 2.1: Aceleración en función del número de procesadores y del porcentaje de paralelización del código.

Se observa que la aceleración es muy sensitiva al porcentaje de paralelización y al número de procesadores. Además, S no crece linealmente conforme el número de procesadores se incrementa. La recomendación es paralelizar códigos cuyo porcentaje paralelizable sea mayor al 70 %.

Escalabilidad

En general un número alto de procesadores disminuye la eficiencia. Por otro lado, se ha observado que en algunos casos el incremento del tamaño del problema incrementa la

eficiencia. Entonces debería ser posible mantener la eficiencia constante cuando, tanto el número de procesadores así como el tamaño del problema se incrementan simultáneamente. Esta habilidad de mantener la eficiencia constante se conoce como escalabilidad. La escalabilidad de un programa paralelo es la medida de su capacidad de incrementar la aceleración en proporción al número de procesadores. Se dice que un programa es escalable cuando su eficiencia es mayor a 0.5.

Capítulo 3

Planteamiento de las ecuaciones de la mecánica de fluidos

Las ecuaciones de la mecánica de fluidos representan leyes de conservación o balance de cantidades físicas como la masa, cantidad de movimiento (*momentum*) y energía. Para obtener las ecuaciones, el fluido se considera continuo y se hace un balance de las diferentes cantidades sobre un elemento de fluido [3, 12]. En este trabajo se desarrolló un sistema para resolver las ecuaciones gobernantes de fluidos newtonianos e incompresibles. En forma general estas ecuaciones se escriben como sigue:

$$\frac{\partial \phi}{\partial t} + \frac{\partial}{\partial x_j}\left(u_j \phi\right) = \frac{\partial}{\partial x_j}\left(\Gamma \frac{\partial \phi}{\partial x_j}\right) + S, \qquad \text{para } j = 1, 2, 3, \tag{3.1}$$

donde ϕ puede representar a la temperatura (T), la densidad (ρ) o a alguna componente de la velocidad ($(u_1, u_2, u_3) \equiv (u, v, w)$); Γ es el coeficiente de difusión y S es el término fuente, que contiene el gradiente presiones y términos provenientes de las fuerzas de cuerpo. En esta notación x_j representa coordenadas cartesianas para la posición, y se utiliza la convención de Einstein en donde índices repetidos se suman.

La ecuación (3.1) se conoce como la ecuación de transporte para la propiedad ϕ. En esta ecuación se distingue claramente el término temporal y el término de convección en el lado izquierdo, y el término difusivo y el término fuente en el lado derecho. La ecuación general de transporte (3.1) es usada como punto de partida para desarrollar algunos métodos numéricos, especialmente el método de volumen finito. En la actualidad existe una gran variedad de técnicas numéricas que permiten resolver este tipo de ecuaciones [19, 28, 58, 84]. En este trabajo se utilizó el método de volumen finito para discretizar las ecuaciones. En este método, al igual que en muchos otros, se requiere de la previa construcción de una malla sobre el dominio de estudio. Para delimitar el alcance de este trabajo, nos enfocaremos en problemas cuyo dominio es rectangular y utilizaremos mallas estructuradas ortogonales. Con el objeto de fijar ideas y para describir con claridad la metodología que usaremos, se tomará como base el problema de convección natural en cavidades. Este problema servirá también para realizar pruebas y comparaciones con resultados de otros autores.

3.1. Convección natural

Las ecuaciones que gobiernan esté fenómeno se obtienen tomando en cuenta la aproximación de Boussinesq [12], es decir la densidad se considera constante excepto en el término de fuerzas de cuerpo, y las propiedades físicas restantes del material se consideran constantes siempre. Entonces, para fluidos newtonianos e incompresibles, las ecuaciones se escriben como sigue:

$$\frac{\partial u_j}{\partial x_j} = 0, \tag{3.2}$$

$$\rho_0 \left[\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right] = -\frac{\partial p}{\partial x_i} + \mu \frac{\partial^2 u_i}{\partial x_j \partial x_j} + \rho b_i, \tag{3.3}$$

$$\frac{\partial T}{\partial t} + u_j \frac{\partial T}{\partial x_j} = \alpha \frac{\partial^2 T}{\partial x_j \partial x_j}, \tag{3.4}$$

donde ρ es la densidad, ρ_0 es una densidad de referencia, μ es la viscosidad dinámica y α es la difusividad térmica. En la ecuación (3.4) se ha considerado que la energía interna se puede escribir como $c_V T$, con c_V el calor específico a volumen constante. También se utiliza la ley de conducción de calor de Fourier, es decir $q_j = -\kappa \frac{\partial T}{\partial x_j}$, siendo κ el coeficiente de conductividad térmica.

Las ecuaciones (3.2), (3.3) y (3.4), se deben complementar con una ecuación de estado para poder ser resueltas. En este trabajo se considera la siguiente relación:

$$\rho = \rho_0 \left[1 - \beta (T - T_0) \right], \tag{3.5}$$

donde β es el coeficiente de expansión volumétrica y T_0 es el valor de la temperatura cuando $\rho = \rho_0$. β se define como:

$$\beta = -\frac{1}{\rho_0} \left(\frac{\partial \rho}{\partial T} \right)_{T=T_0}. \tag{3.6}$$

Esta última ecuación describe satisfactoriamente la relación entre la densidad y la temperatura para un fluido incompresible.

3.1.1. Ecuaciones adimensionales

Las ecuaciones (3.2), (3.3) y (3.4) generalmente se utilizan en forma adimensional tanto para su discretización, como para su implementación. En la forma adimensional, aparecen parámetros que permiten hacer estudios de flujos en diferentes estados: laminar o turbulento por ejemplo. Por esta razón, en este trabajo se utilizan ecuaciones adimensionales, las cuales se obtienen mediante el siguiente escalamiento:

$$x_j = \frac{x'_j}{H}, \tag{3.7}$$

$$t = \frac{t'}{H^2/\alpha}, \tag{3.8}$$

$$u_j = \frac{u_j'}{\alpha/H}, \tag{3.9}$$

$$p = \frac{p'}{\alpha\nu\rho_0/H^2}, \tag{3.10}$$

$$T = \frac{T' - T_C}{\Delta T} - \frac{1}{2}, \tag{3.11}$$

donde localmente se han utilizado variables primadas ($'$) para representar a las variables con dimensiones. En este escalamiento H es una longitud característica, ν la viscosidad cinemática ($\nu = \mu/\rho_0$) y $\Delta T = T_H - T_C$ representa una diferencia de temperaturas.

Usando el escalamiento anterior tenemos que las ecuaciones, en forma adimensional se escriben como sigue:

$$\frac{\partial u_j}{\partial x_j} = 0, \tag{3.12}$$

$$\frac{\partial u_i}{\partial t} + u_j\frac{\partial u_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \mathrm{Pr}\frac{\partial^2 u_i}{\partial x_j \partial x_j} + b_i, \tag{3.13}$$

$$\frac{\partial T}{\partial t} + u_j\frac{\partial T}{\partial x_j} = \frac{\partial^2 T}{\partial x_j \partial x_j}, \tag{3.14}$$

donde la ecuación (3.12) se conoce como la ecuación de continuidad, las ecuaciones (3.13) son las ecuaciones de Navier-Stokes y la ecuación (3.14) es la ecuación de energía. b_i es el término fuente que representa a las fuerzas de cuerpo externas que intervienen en el problema, y en este caso sólo se toma en cuenta el efecto de la gravedad apuntando en la dirección negativa del eje y. Por lo tanto: $b_1 = 0$, $b_2 = \mathrm{Pr}\ \mathrm{Ra}T$ y $b_3 = 0$, donde Pr y Ra son el número de Prandlt y el número de Rayleigh respectivamente que se definen como sigue:

$$\mathrm{Pr} = \frac{\nu}{\alpha}, \tag{3.15}$$

$$\mathrm{Ra} = \frac{g\beta\Delta T L_y^3}{\alpha\nu}, \tag{3.16}$$

donde g es la magnitud de la aceleración de la gravedad.

Las ecuaciones (3.12), (3.13) y (3.14) se pueden escribir en la forma de la ecuación (3.1) como sigue:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_j}\Big(u_j\rho\Big) = 0, \tag{3.17}$$

$$\frac{\partial u_i}{\partial t} + \frac{\partial}{\partial x_j}(u_j u_i) = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left(\mathrm{Pr}\frac{\partial u_i}{\partial x_j}\right) + S_i, \tag{3.18}$$

$$\frac{\partial T}{\partial t} + \frac{\partial}{\partial x_j}(u_j T) = \frac{\partial}{\partial x_j}\left(\frac{\partial T}{\partial x_j}\right). \tag{3.19}$$

Lo anterior es posible dado que el fluido es incompresible y el número de Prandtl se considera constante. En la forma antes escrita, es posible hacer una correspondencia entre las ecuaciones (3.17),(3.18) y (3.19) con la ecuación general, como se muestra en la siguiente tabla.

Ecuación	ϕ	Γ	S_i
Masa	$\rho = \text{cte.}$	0	0
N-S	u_i	Pr	$-\frac{\partial p}{\partial x_i} + b_i$
Energía	T	1	0

Cuadro 3.1: Correspondencia de las ecuaciones de balance en flujo laminar con la ecuación general. El valor de i varía de 1 a 3.

3.2. Convección natural turbulenta

Aunque no existe una definición exacta de turbulencia, se puede decir que un flujo es turbulento cuando es irregular, consiste de un amplio rango de escalas de movimiento, se incrementa la difusividad, es completamente tridimensional, es muy disipativo y el número de Reynolds es relativamente grande. Una discusión amplia del tema se encuentra en [72]. La simulación directa de flujos turbulentos o DNS (*Direct Numerical Simulation*), es posible sólo en algunos casos simples, pero en la mayoría de las ocasiones se requieren recursos enormes de cómputo [41]. Algunas técnicas se han desarrollado para evitar este problema y una de las más conocidas es la Simulación de Vórtices Grandes o LES (*Large-Eddy Simulation*). Los detalles de esta técnica se pueden encontrar en libros de texto como en [18, 84] y en artículos de revisión como [73]. Aquí sólo describiremos las propiedades más importantes.

La técnica de LES ayuda a simular flujos turbulentos en mallas gruesas. La base de esta técnica consiste en la aplicación de un filtro de convolución espacial a las ecuaciones gobernantes. En este procedimiento se divide la variable turbulenta f (T, u_i o p) en una componente de escalas grandes $\bar{f}$, y en una componente de escalas pequeñas o de submalla f'. La descomposición y la convolución de f con una función de filtro g sobre el dominio del flujo Ω se escriben como:

$$f(x_i, t) = \bar{f}(x_i, t) + f'(x_i, t), \tag{3.20}$$

con

$$\bar{f}(x_i,t) = \int_\Omega g(x_i - x_i')f(x_i',t)dx_i'. \tag{3.21}$$

donde la función del filtro g debe satisfacer la condición de normalización

$$\int_\Omega g(x_i - x_i')dx_i' = 1. \tag{3.22}$$

La aplicación del filtro a las ecuaciones gobernantes del problema de convección natural, ecuaciones (3.12),(3.13) y (3.14), produce la siguiente descripción del movimiento de las escalas grandes:

$$\frac{\partial \bar{u}_j}{\partial x_j} = 0, \tag{3.23}$$

$$\frac{\partial \bar{u}_i}{\partial t} + \bar{u}_j\frac{\partial \bar{u}_i}{\partial x_j} = -\frac{\partial \bar{p}}{\partial x_i} + \mathrm{Pr}\frac{\partial^2 \bar{u}_i}{\partial x_j \partial x_j} + \bar{b}_i + \frac{\partial \tau_{ij}}{\partial x_j}, \tag{3.24}$$

$$\frac{\partial \bar{T}}{\partial t} + \bar{u}_j\frac{\partial \bar{T}}{\partial x_j} = \frac{\partial^2 \bar{T}}{\partial x_j \partial x_j} + \frac{\partial h_j}{\partial x_j}. \tag{3.25}$$

los tensores de submalla τ_{ij} y h_j están dados por

$$\tau_{ij} = \bar{u}_i\bar{u}_j - \overline{u_i u_j}, \tag{3.26}$$

y

$$h_j = \bar{u}_j\bar{T} - \overline{u_j T}. \tag{3.27}$$

En la LES se requiere un modelo de submalla o SGM (*Subgrid Model*) que parametrize ambos tensores adecuadamente, de tal manera que el flujo de escalas grandes pueda ser calculado con exactitud. Las simulaciones con el modelo deben producir resultados con un significado físico y con bajo esfuerzo computacional. El SGM más común supone una viscosidad turbulenta (hipótesis de Boussinesq) para modelar τ_{ij}:

$$\tau_{ij} = 2\nu_t\bar{S}_{ij}, \tag{3.28}$$

donde

$$\bar{S}_{ij} = \frac{1}{2}\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right), \tag{3.29}$$

es el tensor de deformación del campo filtrado y ν_t es la viscosidad turbulenta. De igual manera tenemos que

$$h_j = \kappa_t\frac{\partial \bar{T}}{\partial x_j}, \tag{3.30}$$

donde la razón entre la viscosidad turbulenta ν_t y la difusividad turbulenta κ_t está definida como

$$\mathrm{Pr}_t = \frac{\nu_t}{\alpha_t}. \tag{3.31}$$

Cuando el número de Prandtl está dado, solamente la viscosidad turbulenta tiene que ser parametrizada en términos de las cantidades resueltas. Aunque el valor de Pr_t no está bien establecido [2], frecuentemente se considera que $\frac{1}{3} < \mathrm{Pr}_t < \frac{1}{2}$, una discusión sobre este punto se encuentra en [17].

En este trabajo consideraremos el SGM propuesto por Métais y Lesieur [48], en donde la viscosidad turbulenta se define como

$$\nu_t = 0.105 C_k^{-3/2} \Delta \sqrt{\bar{F}_2}, \tag{3.32}$$

con una función de estructura filtrada definida como

$$\bar{F}_2(\vec{x}, \Delta c) = \frac{1}{6} \sum_{i=1}^{3} \bar{F}_2^{(i)} \left(\frac{\Delta c}{\Delta x_i} \right)^{2/3}, \tag{3.33}$$

con

$$\bar{F}_2^{(i)} = \left[\|\vec{u}(\vec{x}) - \vec{u}(\vec{x} + \Delta x_i \vec{e}_i)\|^2 + \|\vec{u}(\vec{x}) - \vec{u}(\vec{x} - \Delta x_i \vec{e}_i)\|^2 \right], \tag{3.34}$$

donde $\vec{e}_i$ es el vector unitario en dirección x_i y $\Delta c = (\Delta x_1 \Delta x_2 \Delta x_3)^{1/3}$.

Sustituyendo (3.28) y (3.30) en las ecuaciones filtradas (3.24) y (3.25), y recordando que el fluido es incompresible, obtenemos el siguiente sistema de ecuaciones:

$$\frac{\partial \bar{u}_i}{\partial t} + \frac{\partial}{\partial x_j} (\bar{u}_j \bar{u}_i) = -\frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j} \Big((\mathrm{Pr} + \nu_t) 2 \bar{S}_{ij} \Big) + \bar{b}_i, \tag{3.35}$$

$$\frac{\partial \bar{T}}{\partial t} + \frac{\partial}{\partial x_j} (\bar{u}_j \bar{T}) = \frac{\partial}{\partial x_j} \left(\left(1 + \frac{\nu_t}{\mathrm{Pr}_t} \right) \frac{\partial \bar{T}}{\partial x_j} \right). \tag{3.36}$$

3.2.1. Función de estructura selectiva

En algunos casos el modelo de la función de estructura es muy disipativo. Para reducir esta dificultad, se han desarrollado varias técnicas entre las que se encuentra el modelo de función de estructura selectiva [48]. El objetivo es eliminar la viscosidad turbulenta cuando el flujo no es lo suficientemente tridimensional. El criterio que se utiliza es como sigue:

1. Se mide el ángulo entre la vorticidad en un punto de la malla y el promedio de las vorticidades en los seis puntos vecinos.

2. Si este ángulo excede 20^o entonces se toma en cuenta la viscosidad turbulenta calculada mediante (3.32).

3. En otro caso, sólo se toma en cuenta la viscosidad molecular.

El valor de 20^o es el más probable de acuerdo a simulaciones de turbulencia isotrópica a una resolución de malla de 32^3 y 64^3 [48].

De la misma manera en que las ecuaciones del caso laminar se escribieron en forma general, las ecuaciones (3.23), (3.35) y (3.36) tambien están escritas en la forma (3.1). En este caso la correspondencia con la ecuación general es como sigue:

Ecuación	ϕ	Γ	S_i
Masa	$\bar{\rho} = \text{cte}$	0	0
N-S	$\bar{u}_i$	$\text{Pr} + \nu_t$	$-\frac{\partial \bar{p}}{\partial x_i} + \bar{b}_i$
Energía	T	$1 + \frac{\nu_t}{\text{Pr}_t}$	0

Cuadro 3.2: Correspondencia de las ecuaciones de balance en flujo turbulento con la ecuación general. El valor de i varía de 1 a 3.

3.3. Discusión

Las ecuaciones presentadas en las secciones anteriores son bastante complejas y a pesar de que se conocen desde hace mucho tiempo, no existe hoy en día un método analítico para encontrar sus soluciones. Por esta razón, se han desarrollado muchos métodos numéricos con los que es posible encontrar soluciones numéricas aproximadas. Mediante el uso de las arquitecturas sofisticadas de cómputo y en especial de las arquitecturas multiprocesador, ha sido posible resolver las ecuaciones gobernantes para diferentes tipos de flujos, con excelente exactitud y precisión.

Sin embargo, debido a la complejidad de los problemas que se desean resolver numéricamente, se desarrollan programas especializados para cada tipo de flujo. Generalmente lo que se busca es resolver los sistemas de ecuaciones de una manera eficiente. No obstante, cuando se requiere modificar el programa para usar otras técnicas de solución (como esquemas numéricos diferentes), es bastante complicado y es preferible iniciar un programa nuevo desde el principio. Esto último es un caso típico de lo que se conoce como la crisis del software, que es la incapacidad de desarrollar software que sea los suficientemente simple para ser entendido, mantenido y extendido por cualquier programador, y que además sea capaz de proveer soluciones a problemas muy complejos, véase por ejemplo [7]. El problema de la crisis del software, surge en aplicaciones de cómputo numérico debido a que, la construcción de este tipo software no se hace siguiendo alguna técnica ordenada de desarrollo de software, como la descrita en [38], y sólo se pone atención en la eficiencia.

Como es sabido, es posible resolver el mismo problema usando diferentes técnicas numéricas y en muchos casos es de interés realizar una comparación entre cada una de ellas para determinar cual funciona mejor para un problema particular. Entonces, la

flexibilidad del código, además de la eficiencia, es importante en la solución de problemas numéricos. Esta es una de las razones por la que se utilizan procesos de desarrollo ordenados y paradigmas de programación modernos para el desarrollo de nuestro sistema.

Capítulo 4

Método numérico

En general, las ecuaciones de balance (3.12)–(3.14) no pueden ser resueltas en forma analítica. En la actualidad, existen diferentes estratégias numéricas con las que se pueden obtener soluciones numéricas aproximadas. Para esto se han propuesto métodos como diferencias finitas, elemento finito, volumen finito, métodos espectrales, etc. Estos métodos se encuentran descritos en [28]. El método de Volumen Finito ha sido utilizado con éxito en muchas simulaciones de dinámica de fluidos y su efectividad se debe principalmente a que es un método conservativo, es decir, se aplica el principio de conservación en cada volumen de control de la malla para obtener las ecuaciones discretas. En las secciones siguientes se describe brevemente la forma en que se aplica el método de Volumen Finito a las ecuaciones de balance para obtener su forma discreta. Para más detalles del método se recomienda revisar [58, 84]. La descripción del método se hace en términos de la ecuación general (3.1) descrita en el capítulo anterior.

4.1. Volumen Finito

La idea básica del método de volumen finito o volumen de control, es fácil de entender y permite una interpretación física directa. Primero, el dominio de estudio se divide en un número de volúmenes de control no sobrepuestos, de tal manera que existe un volumen rodeando a cada punto de la malla, como se muestra en la figura 4.1. En el esquema que usaremos se agregan puntos sobre la frontera del dominio para tomar en cuenta las condiciones de frontera. La ecuación general (3.1) se integra sobre cada uno de los volúmenes de control y se utilizan diferentes aproximaciones o esquemas numéricos, para cada uno de los términos resultantes. De esta manera se obtiene un conjunto de ecuaciones discretas, una para cada volumen de la malla, en donde la variable escalar ϕ (que puede representar a T, ρ, o a alguna componente de la velocidad) se escribe en términos de valores en puntos vecinos de esa misma variable. Las ecuaciones discretas obtenidas de esta forma, expresan el principio de conservación para ϕ en cada volumen de control, de la misma forma que la ecuación diferencial expresa el mismo principio para un volumen de fluido infinitesimal. Esta formulación implica que la solución resultante debe cumplir con el principio de conservación de masa, de cantidad de movimiento y

de energía, sobre cualquier grupo de volúmenes de control y por lo tanto, sobre todo el dominio. Esta característica se cumple independientemente del número de puntos de la malla y no sólo cuando la malla es excesivamente fina. Entonces, aun para una malla gruesa la solución exhibirá un comportamiento cualitativamente realista, aunque en estos casos no se resolverán los detalles dinámicos finos y la solución tendrá menos precisión.

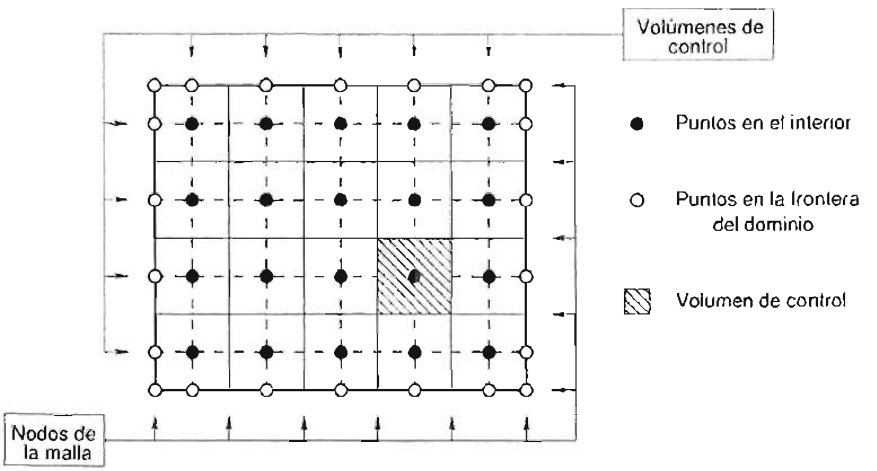


Figura 4.1: Dominio de estudio discretizado usando volúmenes de control en 2D.

4.1.1. Discretización de la ecuación general

Considérese una malla rectangular estructurada [1] en tres dimensiones. Los volúmenes de control en 3D sobre los que se integra la ecuación (3.1) son como el que se muestra en la figura 4.2. Los subíndices E, W, N, S, F y B indican los puntos vecinos al punto P de la malla, mientras que las etiquetas e, w, n, s, f y b indican las caras del volumen de control. La integración, espacial y temporal de la ecuación (3.1) produce la siguiente expresión:

$$(\phi - \phi^{\circ})\frac{\Delta V}{\Delta t} + \mathrm{C} = \mathrm{D} + \mathrm{S}, \tag{4.1}$$

donde ϕ es el valor de la variable escalar en el tiempo $t + \Delta t$, mientras que ϕ° representa el valor en el mismo punto pero al tiempo t. La integración en el tiempo se realiza en el intervalo Δt. La forma de los términos C, D y S, convectivo, difusivo y fuente respectivamente, dependen del esquema numérico utilizado en la discretización de cada uno de ellos. En este trabajo se utiliza un esquema implícito (Backward Euler), haciendo que los términos C, D y S sean funciones de ϕ. Otros esquemas como el explícito (Forward Euler) o el de Crank-Nicolson pueden implementarse fácilmente [58]. Utilizando

[1] En una malla estructurada todos los puntos tienen el mismo número de vecinos, a excepción de los puntos en la frontera [20].

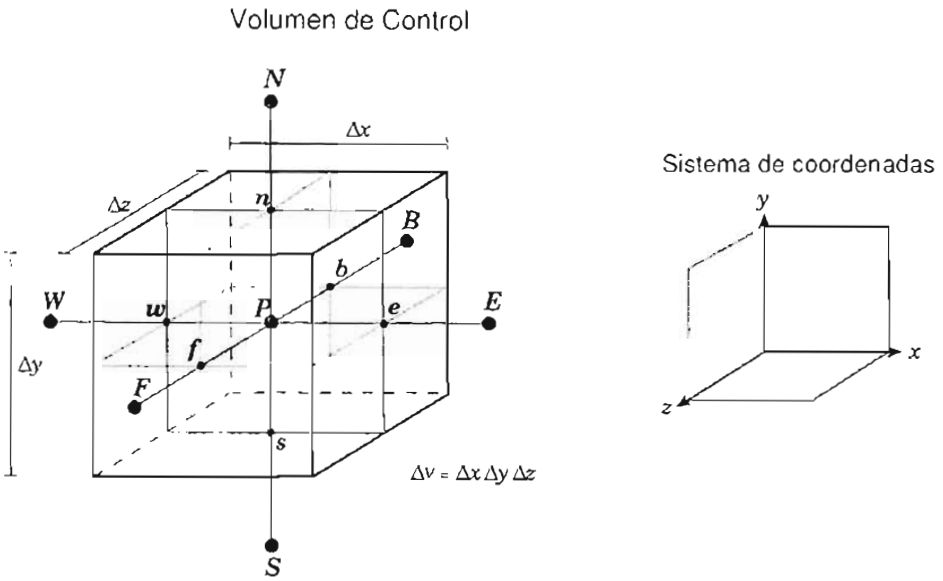


Figura 4.2: Volumen de control alrededor del punto P.

la nomenclatura de la figura 4.2, la forma general de los términos de la ecuación (4.1) es como sigue:

$$\mathrm{C} = (c_e\phi_e - c_w\phi_w) + (c_n\phi_n - c_s\phi_s) + (c_f\phi_f - c_b\phi_b), \tag{4.2}$$

$$\begin{aligned}\mathrm{D} = \Gamma\left[\left(\frac{\partial\phi}{\partial x}\right)_e - \left(\frac{\partial\phi}{\partial x}\right)_w\right]\Delta y\Delta z \\ + \Gamma\left[\left(\frac{\partial\phi}{\partial y}\right)_n - \left(\frac{\partial\phi}{\partial y}\right)_s\right]\Delta x\Delta z \\ + \Gamma\left[\left(\frac{\partial\phi}{\partial z}\right)_f - \left(\frac{\partial\phi}{\partial z}\right)_b\right]\Delta x\Delta y,\end{aligned} \tag{4.3}$$

$$\mathrm{S} = S_p\Delta V, \tag{4.4}$$

donde los términos c de la ecuación (4.2) están definidos de la siguiente manera:

$$\begin{aligned} c_e &= u_e\Delta y\Delta z, \quad c_w = u_w\Delta y\Delta z, \\ c_n &= v_n\Delta x\Delta z, \quad c_s = v_s\Delta x\Delta z, \\ c_f &= w_f\Delta x\Delta y, \quad c_b = w_b\Delta x\Delta y, \end{aligned} \tag{4.5}$$

Los términos convectivos y difusivos se pueden aproximar usando diferentes esquemas [84]. Independientemente de la aproximación usada, cuando se insertan estos es-

quemas en las ecuaciones (4.2) y (4.3), y éstos a su vez en la ecuación (4.1), se obtienen sistemas lineales como el siguiente:

$$a_P\phi_P = a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_S + a_F\phi_F + a_B\phi_B + S_P, \tag{4.6}$$

donde los coeficientes contienen una parte difusiva y otra convectiva:

$$\begin{array}{llll} a_E = D_E + C_E, & & a_W = D_W + C_W, \\ a_N = D_N + C_N, & & a_S = D_S + C_S, \\ a_F = D_F + C_F, & & a_B = D_B + C_B, \\ a_P = D_P + C_P + \frac{\Delta V}{\Delta t}. & & \end{array} \tag{4.7}$$

En este trabajo se utilizaron diferentes aproximaciones para calcular numéricamente los términos convectivos y difusivos, definidos en las ecuaciones (4.2) y (4.3). En las secciones que siguen se hace una descripción detallada de dichas aproximaciones.

4.2. Aproximación de los términos difusivos

Las derivadas parciales que aparecen en la ecuación (4.3), se deben evaluar en las caras del volumen de control. Estas derivadas pueden aproximarse usando un perfil lineal entre puntos adyacentes de la malla (p. ej. entre P y E). De esta manera tenemos que:

$$\begin{array}{llll} \left(\frac{\partial\phi}{\partial x}\right)_e \simeq \frac{\phi_E - \phi_P}{\Delta x_e}, & & \left(\frac{\partial\phi}{\partial x}\right)_w \simeq \frac{\phi_P - \phi_W}{\Delta x_w}, \\ \left(\frac{\partial\phi}{\partial y}\right)_n \simeq \frac{\phi_N - \phi_P}{\Delta y_n}, & & \left(\frac{\partial\phi}{\partial y}\right)_s \simeq \frac{\phi_P - \phi_S}{\Delta y_s}, \\ \left(\frac{\partial\phi}{\partial z}\right)_f \simeq \frac{\phi_F - \phi_P}{\Delta z_f}, & & \left(\frac{\partial\phi}{\partial z}\right)_b \simeq \frac{\phi_P - \phi_B}{\Delta z_b}. \end{array} \tag{4.8}$$

donde $\Delta x_e, \Delta x_w, \Delta y_n, \Delta y_s, \Delta z_f$ y Δz_b, son definidos como se muestra en la figura 4.3.

Cuando la malla es uniforme ($\Delta x_e = \Delta x_w = \Delta x$, idénticamente en las direcciones y y z), se puede demostrar, a partir de la expansión en series de Taylor de ϕ_E, ϕ_W, ϕ_N, ϕ_S, ϕ_F y ϕ_B, que las expresiones (4.8) producen una aproximación de orden $\mathcal{O}(\Delta x^2)$, [84]. Con estas aproximaciones la parte difusiva de los coeficientes (4.7) es:

$$\begin{array}{llll} D_E = \Gamma\frac{\Delta y\Delta z}{\Delta x_e}, & & D_W = \Gamma\frac{\Delta y\Delta z}{\Delta x_w}, \\ D_N = \Gamma\frac{\Delta x\Delta z}{\Delta y_n}, & & D_S = \Gamma\frac{\Delta x\Delta z}{\Delta y_s}, \\ D_F = \Gamma\frac{\Delta x\Delta y}{\Delta z_f}, & & D_B = \Gamma\frac{\Delta x\Delta y}{\Delta z_b}, \\ D_P = D_E + D_W + D_N & & +D_S + D_F + D_B. \end{array} \tag{4.9}$$

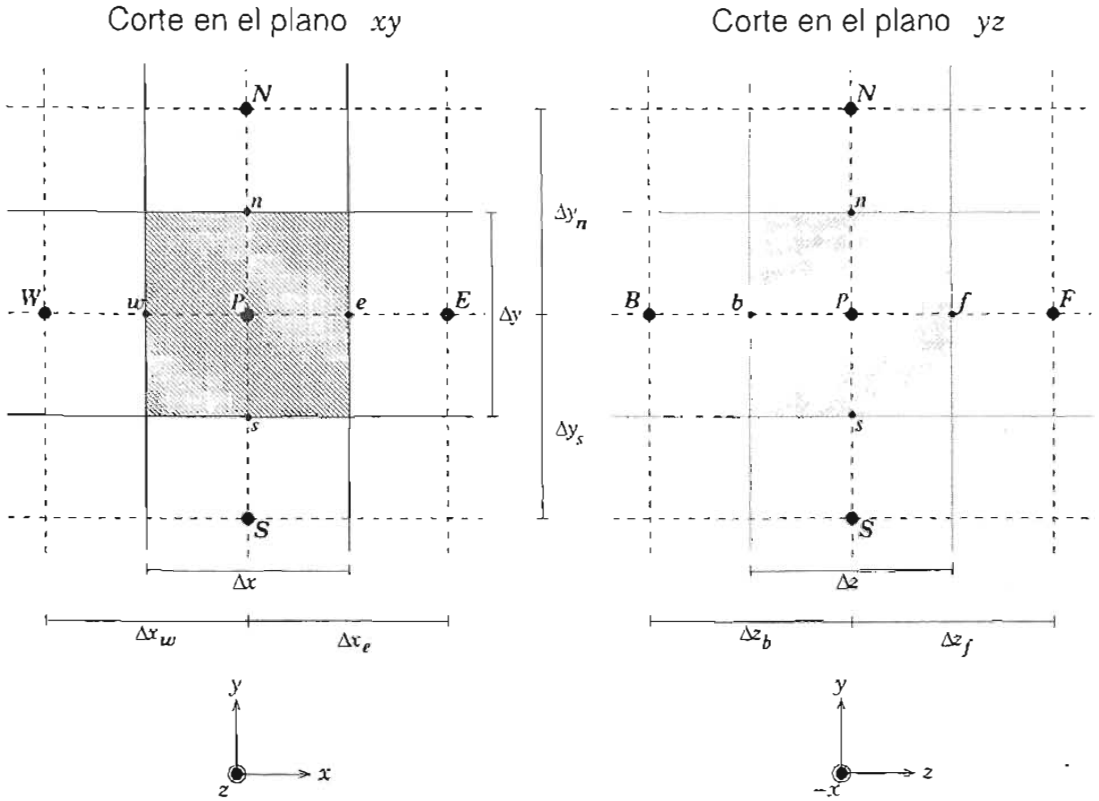


Figura 4.3: Cortes del dominio discreto en los planos xy y yz.

Esta discretización para los términos difusivos es la única que se utiliza en todos los ejemplos de este trabajo.

4.3. Aproximación de los términos convectivos

Los términos convectivos son importantes dado que representan la parte no lineal de la ecuación general (3.1). En la ecuación (4.2) se observa que es necesario encontrar los valores de ϕ en las caras del volumen de control. Sin embargo, ϕ representa a una variable escalar definida en los centros de los volúmenes, como se muestra en las figuras 4.2 y 4.3. En este trabajo se implementaron tres diferentes esquemas para aproximar ϕ en las caras de los volúmenes: Upwind, diferencias centrales (CDS) y QUICK.

4.3.1. Upwind

Este es un esquema que proporciona una aproximación de primer orden $\mathcal{O}(\Delta x)$, y consiste en tomar el valor de la variable escalar en la cara del volumen de control, igual

al valor de ϕ en el punto de la malla de donde proviene el flujo (*upstream*). Por ejemplo, si $c_e > 0 \Longrightarrow \phi_e = \phi_P$ y si $c_w > 0 \Longrightarrow \phi_w = \phi_W$, la figura 4.4 muestra este caso.

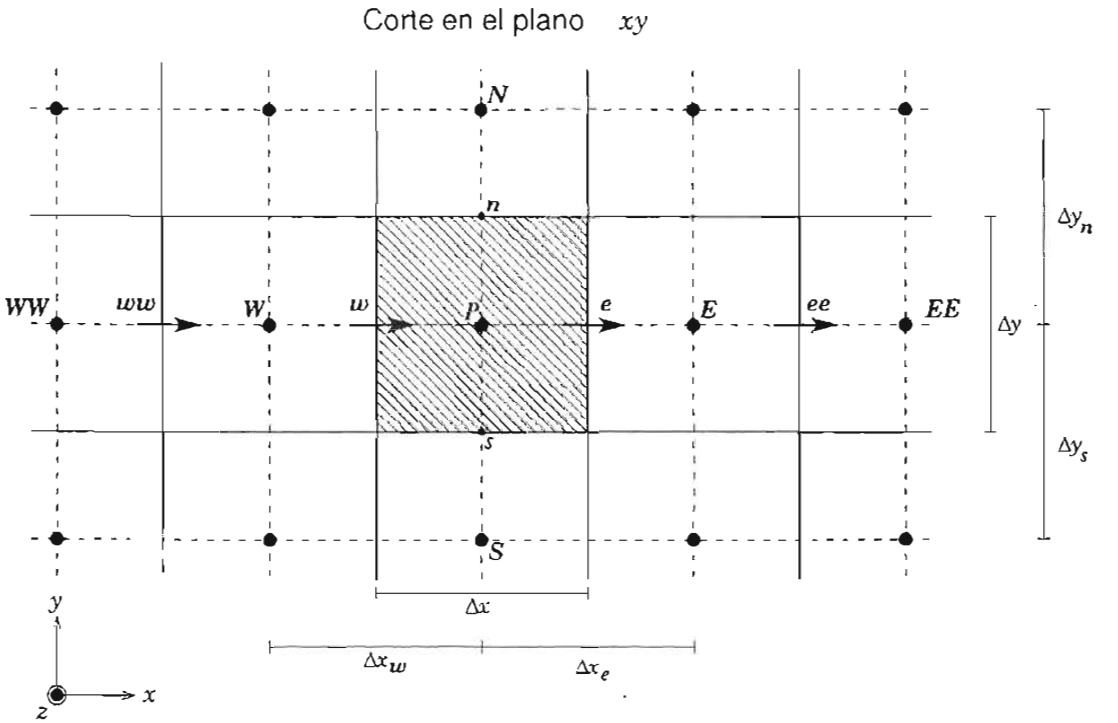


Figura 4.4: Esquema Upwind para el caso: $c_e > 0 \Longrightarrow \phi_e = \phi_P$ y $c_w > 0 \Longrightarrow \phi_w = \phi_W$.

Las funciones que definen este esquema para ϕ_e y ϕ_w, son:

$$\phi_e = \begin{cases} \phi_P & \text{si} \quad c_e > 0, \\ \phi_E & \text{si} \quad c_e \leq 0, \end{cases} \qquad \text{y} \qquad \phi_w = \begin{cases} \phi_W & \text{si} \quad c_w > 0, \\ \phi_P & \text{si} \quad c_w \leq 0. \end{cases} \tag{4.10}$$

Funciones similares se definen para ϕ_n, ϕ_s, ϕ_f y ϕ_b. Ahora, si se define $[|A, B|]$ como el máximo entre A y B, entonces, los términos convectivos en este esquema se expresan de la siguiente manera:

$$\begin{aligned} c_e\phi_e &= \Big(\phi_P\,[|c_e, 0|] - \phi_E\,[|-c_e, 0|]\Big), & c_w\phi_w &= \Big(\phi_W\,[|c_w, 0|] - \phi_P\,[|-c_w, 0|]\Big), \\ c_n\phi_n &= \Big(\phi_P\,[|c_n, 0|] - \phi_N\,[|-c_n, 0|]\Big), & c_s\phi_s &= \Big(\phi_S\,[|c_s, 0|] - \phi_P\,[|-c_s, 0|]\Big), \\ c_f\phi_f &= \Big(\phi_P\,[|c_e, 0|] - \phi_F\,[|-c_e, 0|]\Big), & c_b\phi_b &= \Big(\phi_B\,[|c_b, 0|] - \phi_P\,[|-c_b, 0|]\Big). \end{aligned} \tag{4.11}$$

Sustituyendo estas definiciones en (4.2) obtenemos la siguiente forma para la parte convectiva de los coeficientes (4.7):

$$\begin{aligned} C_E &= [|-c_e, 0|], \quad C_W = [|c_w, 0|], \quad C_N = [|-c_n, 0|], \\ C_S &= [|c_s, 0|], \quad C_F = [|-c_f, 0|], \quad C_B = [|c_b, 0|]. \\ C_P &= C_E + C_W + C_N + C_S + C_F + C_B + (c_e - c_w) + (c_n - c_s) + (c_f - c_b). \end{aligned} \tag{4.12}$$

Como se verá más adelante, la expresión $(c_e - c_w) + (c_n - c_s) + (c_f - c_b)$, es la versión discreta de la ecuación de continuidad.

4.3.2. Diferencias centrales (CDS)

Una manera simple de evaluar ϕ en las caras, es mediante una interpolación lineal usando valores de puntos vecinos. Por ejemplo, en la cara e tenemos que:

$$\phi_e = \phi_E \lambda_e + \phi_P (1 - \lambda_e), \tag{4.13}$$

donde el factor de interpolación λ_e se define como:

$$\lambda_e = \frac{x_e - x_P}{x_E - x_P}. \tag{4.14}$$

Cuando la malla es uniforme $\lambda_e = 0.5$. La precisión de la ecuación (4.13) es de segundo orden ($\mathcal{O}(\Delta x^2)$) como puede mostrase a través de una expansión en series de Taylor de ϕ_E alrededor del punto P. Este es el esquema de segundo orden más simple y corresponde a la aproximación de diferencias centrales de la primera derivada en los métodos de diferencias finitas.

Con este esquema los coeficientes son de la forma:

$$\begin{aligned} C_E &= -c_e \lambda_e, \quad C_W = c_e \lambda_w, \quad C_N = -c_e \lambda_n, \\ C_S &= c_e \lambda_s, \quad C_F = -c_e \lambda_f, \quad C_B = c_e \lambda_b, \\ C_P &= C_E + C_W + C_N + C_S + C_F + C_B + (c_e - c_w) + (c_n - c_s) + (c_f - c_b). \end{aligned} \tag{4.15}$$

4.3.3. QUICK

El esquema QUICK (*Quadratic Upstream Interpolation for Convective Kinematics*) desarrollado por Leonard [47] es un esquema tipo Upwind de tercer orden para interpolar flujos convectivos en las caras de los volúmenes de control. En este caso, se seleccionan tres puntos de la malla para construir un polinomio de segundo grado. La selección de los puntos se hace de acuerdo con la dirección de la velocidad en la cara correspondiente del volumen de control. Los detalles de la construcción de los polinomios se describen en el apéndice B.

La fórma cuadrática de interpolación de los términos convectivos $c_e\phi_e$ y $c_w\phi_w$, en el esquema QUICK, para mallas uniformes tiene la siguiente forma:

$$c_e\phi_e = c_e\left(\frac{\phi_P + \phi_E}{2}\right) - c_{ep}\left(\phi_W - 2\phi_P + \phi_E\right) - c_{em}\left(\phi_{EE} - 2\phi_E + \phi_P\right), \quad (4.16)$$

$$c_w\phi_w = c_w\left(\frac{\phi_P + \phi_W}{2}\right) - c_{wp}\left(\phi_{WW} - 2\phi_W + \phi_P\right) - c_{wm}\left(\phi_E - 2\phi_P + \phi_W\right), \quad (4.17)$$

en donde c_{ep}, c_{em}, c_{wp} y c_{wm} son funciones que dependen de c_e y c_w. La forma de c_{ep} y c_{em} se muestra en la ecuación (4.18) y en la figura 4.5; definiciones similares se hacen para c_{wp} y c_{wm}. Los términos $c_n\phi_n$, $c_s\phi_s$, $c_f\phi_f$ y $c_b\phi_b$ se aproximan de igual manera, véase apéndice B.

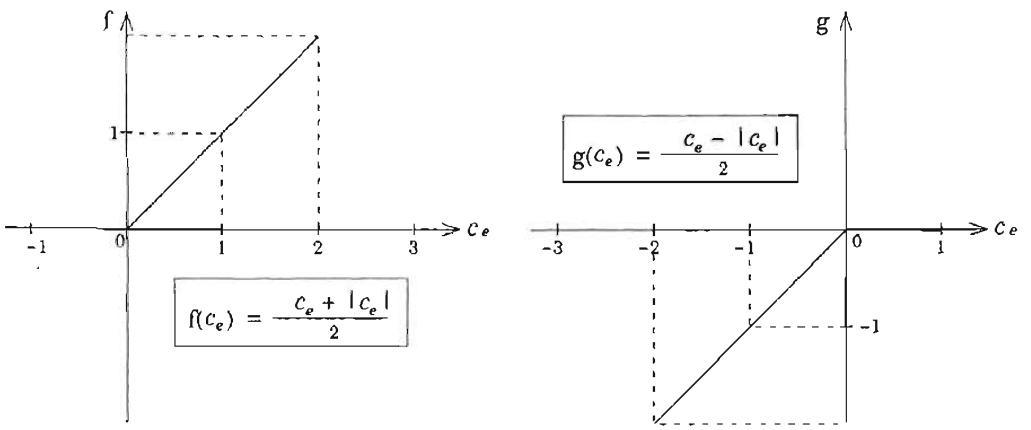


Figura 4.5: Definición de las funciones f y g.

$$c_{ep} = \frac{f}{8} = \begin{cases} \frac{c_e}{8} & \text{si } c_e > 0, \\ 0 & \text{si } c_e \leq 0, \end{cases} \qquad c_{em} = \frac{g}{8} = \begin{cases} 0 & \text{si } c_e \geq 0, \\ -\frac{c_e}{8} & \text{si } c_e < 0. \end{cases} \quad (4.18)$$

Usando las definiciones anteriores, los coeficientes convectivos toman la forma siguiente:

$$\begin{aligned}
C_E &= -\frac{c_e}{2} + c_{ep} - 2c_{em} + c_{wm}, & C_W &= \frac{c_w}{2} + 2c_{wp} - c_{wm} + c_{ep}, \\
C_N &= -\frac{c_n}{2} + c_{np} - 2c_{nm} + c_{sm}, & C_S &= \frac{c_s}{2} + 2c_{sp} - c_{sm} + c_{np}, \\
C_F &= -\frac{c_f}{2} + c_{fp} - 2c_{fm} + c_{bm}, & C_B &= \frac{c_b}{2} + 2c_{bp} - c_{bm} + c_{fp}, \\
C_P &= C_E + C_W + C_N + C_S + C_F + C_B + c_{em} - c_{wp} + c_{nm} - c_{sp} + c_{fm} - c_{bp} + \\
&\quad (c_e - c_w) + (c_n - c_s) + (c_f - c_b).
\end{aligned} \quad (4.19)$$

El orden de aproximación de este esquema es de $\mathcal{O}(\Delta x^3)$, véase [47]. En el apéndice B se describe la forma de los coeficientes para mallas no uniformes.

4.3.4. Condiciones de frontera

La discretización de las condiciones de frontera es importante dado que definen la solución que se obtiene en el interior del dominio de estudio. Para los problemas que trataremos existen básicamente dos tipos de condiciones de frontera:

- Dirichlet: en donde el valor del campo se define en la frontera, es decir: $\phi = \phi_b$.
- Neumann: el gradiente del campo normal a la frontera es especificado, es decir $\partial\phi/\partial n = \phi'_b$.

En la figura 4.6, por ejemplo, el punto E cae fuera del dominio y la cara e del volumen que rodea a P cae justo en la frontera. Usando las técnicas de discretización descritas antes, una ecuación para el punto P en términos de sus vecinos se escribe de forma similar a la ecuación (4.6). Sin embargo, en este caso algunos de los coeficientes cambian su forma debido a que el punto E no está dentro del dominio.

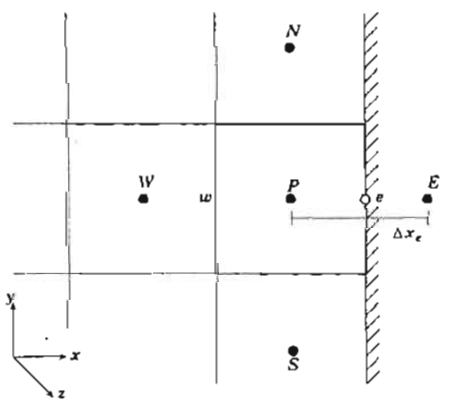


Figura 4.6: Volumen de control en la frontera.

En el caso de la figura 4.6, la condición de frontera de tipo Dirichlet es impuesta de tal manera que $\phi_b = \phi_e$. El valor de la frontera se puede aproximar mediante una interpolación lineal simple:

$$\phi_e = \phi_b \approx \frac{\phi_P + \phi_E}{2}, \tag{4.20}$$

de donde se obtiene:

$$\phi_E = 2\phi_b - \phi_P. \tag{4.21}$$

Sustituyendo esta última expresión en la ecuación (4.6) y factorizando obtenemos:

$$a_P^* \phi_P = a_E^* \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + a_F \phi_F + a_B \phi_B + S_P^*, \tag{4.22}$$

donde

$$\begin{aligned} a_P^* &= a_P + a_E, \\ S_P^* &= S_P + 2a_E \phi_b \quad \text{y} \\ a_E^* &= 0. \end{aligned}$$

Para las condiciones de tipo Neumann, usando diferencias centrales para aproximar el gradiente normal a la superficie, tenemos que:

$$\phi_b' = \left(\frac{\partial \phi}{\partial x} \right)_e \approx \frac{\phi_E - \phi_P}{\Delta x_e}, \tag{4.23}$$

de donde obtenemos:

$$\phi_E = \phi_P + \Delta x_e \phi_b'. \tag{4.24}$$

Sustituyendo esta expresión en la ecuación (4.6) obtenemos una ecuación similar a (4.22), con los coeficientes definidos como sigue:

$$\begin{aligned} a_P^* &= a_P - a_E, \\ S_P^* &= S_P + a_E \Delta x_e \phi_b' \quad \text{y} \\ a_E^* &= 0. \end{aligned}$$

De esta manera, las condiciones de frontera, Dirichlet y Neumann, se incorporan en el sistema de ecuaciones a resolver. Es posible utilizar diferentes aproximaciones para las condiciones de frontera, y esto depende del orden de aproximación del esquema numérico utilizado en los puntos interiores, véase [84].

4.4. Acoplamiento presión-velocidad

La convección de una variable escalar ϕ depende de la magnitud y dirección de la velocidad. En general, la velocidad no se conoce y debe calcularse como parte del proceso de solución junto con las otras variables del flujo. Las ecuaciones para cada componente de la velocidad — ecuaciones de cantidad de movimiento — pueden derivarse de la ecuación general (3.1). En el caso de flujos incompresibles, la velocidad debe satisfacer la ecuación de continuidad (3.12). Para describir el método de solución, reescribimos las ecuaciones de cantidad de movimiento y de continuidad como sigue:

$$\begin{aligned}
\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial v}{\partial y}+w\frac{\partial w}{\partial z} &= -\frac{\partial p}{\partial x}+\Gamma\frac{\partial^2 u}{\partial x^2}+\Gamma\frac{\partial^2 u}{\partial y^2}+\Gamma\frac{\partial^2 u}{\partial z^2}+S_u,\\
\frac{\partial v}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial v}{\partial y}+w\frac{\partial w}{\partial z} &= -\frac{\partial p}{\partial y}+\Gamma\frac{\partial^2 v}{\partial x^2}+\Gamma\frac{\partial^2 v}{\partial y^2}+\Gamma\frac{\partial^2 v}{\partial z^2}+S_v,\\
\frac{\partial w}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial v}{\partial y}+w\frac{\partial w}{\partial z} &= -\frac{\partial p}{\partial z}+\Gamma\frac{\partial^2 w}{\partial x^2}+\Gamma\frac{\partial^2 w}{\partial y^2}+\Gamma\frac{\partial^2 w}{\partial z^2}+S_w.
\end{aligned} \tag{4.25}$$

$$\frac{\partial u}{\partial x}+\frac{\partial u}{\partial x}+\frac{\partial u}{\partial x}=0. \tag{4.26}$$

La solución de estas ecuaciones presenta algunos problemas:

- Los términos convectivos de las ecuaciones de cantidad de movimiento son cantidades no lineales.
- Las ecuaciones están fuertemente acopladas debido a que cada componente de la velocidad aparece en cada ecuación de cantidad de movimiento.
- El problema más complejo es resolver el papel que juega la presión: no existe explícitamente una ecuación para la presión.

Estos problemas, asociados con la no linealidad de las ecuaciones, pueden resolverse adoptando una estrategia de solución iterativa. En este trabajo se utiliza el método SIMPLEC (*Semi-Implicit Method for Pressure Linked Equations - Consistent*) [16] y algunas de sus modificaciones para resolver los problemas planteados en el capítulo 3.

4.4.1. Mallas desplazadas

La forma más sencilla de aproximar el gradiente de presiones, que aparece en las ecuaciones (4.25), es mediante una interpolación lineal. Por ejemplo, para la ecuación en dirección x tenemos, de la figura 4.4, que:

$$\left(\frac{\partial p}{\partial x}\right)_P \approx \frac{p_e-p_w}{\Delta x}=\frac{\left(\frac{p_E+p_P}{2}\right)-\left(\frac{p_P+p_W}{2}\right)}{\Delta x}=\frac{p_E-p_W}{\Delta x}. \tag{4.27}$$

En la ecuación anterior se observa que el valor de la presión en el nodo central P, no aparece. Esto puede ocasionar campos de presiones con oscilaciones no realistas. Es claro entonces, que si las velocidades están definidas en los puntos centrales de la malla, la influencia de la presión no estará representada correctamente. Un remedio a este problema es utilizar mallas desplazadas (*staggered grids*) para las componentes de la velocidad. La idea es evaluar las variables escalares, tales como la presión y la temperatura en los centros de los volúmenes de control; mientras que las velocidades se evaluan en las caras de los mismos volúmenes. El arreglo para las tres componentes

de la velocidad se muestra en la figura 4.7. Se observa que el punto central P, para la componente u de la velocidad, se desplaza a la cara w. Para las otras componentes se hace un desplazamiento similar. Con este arreglo, el gradiente de presiones, en la ecuación para u se calcula de la siguiente forma:

$$\left(\frac{\partial p}{\partial x}\right)_w \approx \frac{p_P - p_W}{\Delta x}. \tag{4.28}$$

Una ventaja adicional de las mallas desplazadas es que genera velocidades en los lugares exactos donde se requiere para las ecuaciones escalares y por lo tanto no es necesario realizar interpolaciones.

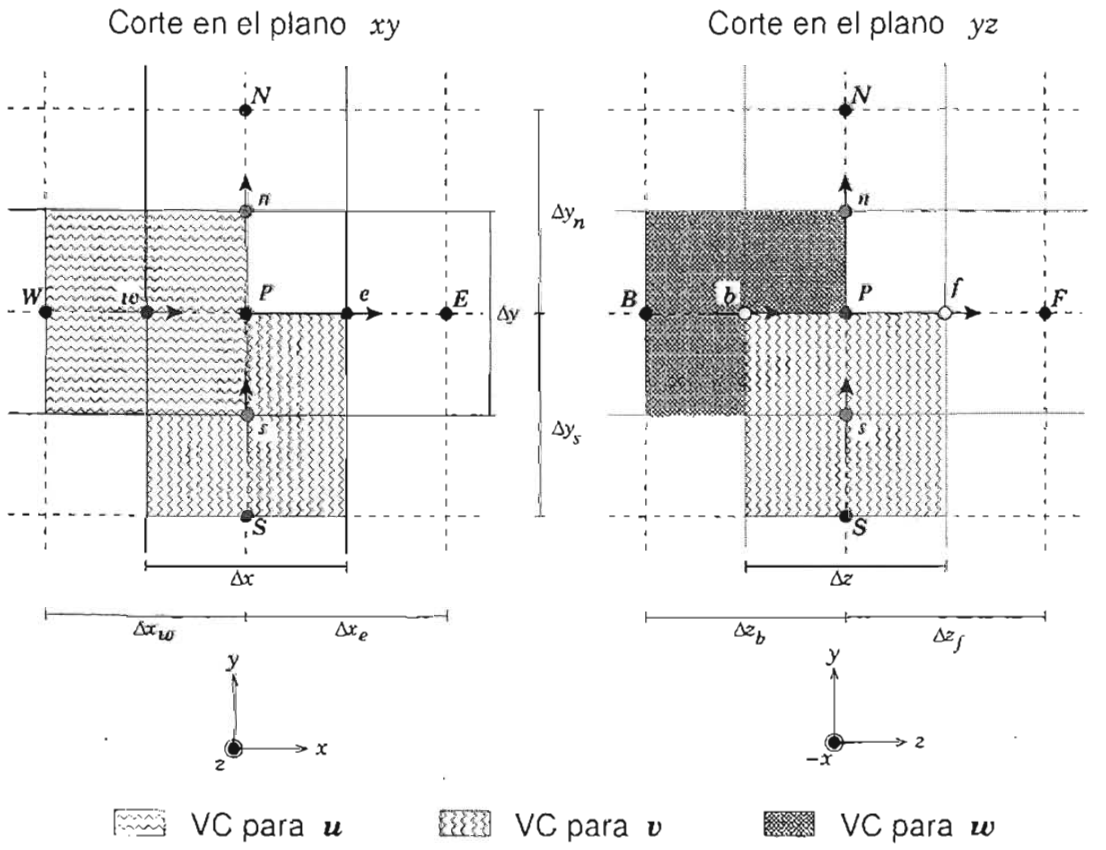


Figura 4.7: Malla y volúmenes de control (VC) para las componentes de la velocidad.

4.4.2. Ecuación de corrección a la presión

Una relación para calcular la presión se puede obtener a partir de las ecuaciones (4.25) y (4.26). Aplicando el método de volumen finito a la ecuación de cantidad de

movimiento para u en una malla desplazada como se muestra en la figura 4.7, obtenemos:

$$a_P u_P = \sum_{nb} a_{nb} u_{nb} + b_u + A_u(p_W - p_P), \tag{4.29}$$

donde se escribe explícitamente el gradiente de presiones. En esta ecuación tenemos que $nb = E, W, N, S, F, B$, $A_u = \Delta y \Delta z$ es el área de la cara w del volumen de control y b_u es el término fuente.

Ahora, dado que no conocemos la presión, definimos p^* como una presión aproximada. Esta presión produce una velocidad aproximada u^*. Por lo tanto tenemos que:

$$a_P u_P^* = \sum_{nb} a_{nb} u_{nb}^* + b_u + A_u(p_W^* - p_P^*). \tag{4.30}$$

Para obtener u y p correctas se deben corregir los valores aproximados mediante $p = p^* + p'$ y $u = u^* + u'$. Una relación entre p' y u' se obtiene restando las ecuaciones (4.29) y (4.30):

$$a_P u_P' = \sum_{nb} a_{nb} u_{nb}' + A_u(p_W' - p_P'). \tag{4.31}$$

En los métodos del tipo SIMPLE se encuentra una ecuación discreta ya sea para la presión p o para la corrección a la presion p' y se resuelve dentro del ciclo global del método. Enseguida describimos el método SIMPLEC que es una modificación del SIMPLE que permite obtener convergencia en menos iteraciones.

4.4.3. SIMPLEC

En el SIMPLE se realiza una aproximación *consistente* que nos lleva a una expresión sencilla para p'. En este algoritmo se resta el término $\sum a_{nb} u_P'$ en ambos lados de la ecuación (4.31), de tal forma que se tiene:

$$(a_P - \sum_{nb} a_{nb}) u_P' = \underbrace{\sum_{nb} a_{nb}(u_{nb}' - u_P')}_{\approx 0} + A_u(p_W' - p_P'). \tag{4.32}$$

La diferencia $(u_{nb}' - u_P')$ es aproximadamente igual a cero, para todo nb y para mallas relativamente finas. Por lo tanto, es posible eliminar este término de la ecuación, de tal manera que la velocidad corregida estará dada por:

$$u_P = u_P^* + u_P' = u_P^* + d_u(p_W' - p_P'), \tag{4.33}$$

donde

$$d_e = A_e/(a_P - \sum a_{nb}). \tag{4.34}$$

Las correciones para las componentes v y w se escriben como:

$$v_P = v_P^* + v_P' = v_P^* + d_v(p_S' - p_P'), \tag{4.35}$$

$$w_P = w_P^* + w_P' = w_P^* + d_w(p_B' - p_P'), \tag{4.36}$$

La ecuación para p' se obtiene sustituyendo las ecuaciones (4.33), (4.35) y (4.36) en la ecuación (4.26) :

$$a_P p_P' = a_E p_E' + a_W p_W' + a_N p_N' + a_S p_S' + a_F p_F' + a_B p_B' + b_p, \tag{4.37}$$

donde los coeficientes están definidos como sigue:

$$\begin{aligned} a_E' &= d_u A_u, \quad a_W' = d_u A_u, \quad a_N' = d_v A_v, \\ a_S' &= d_v A_v, \quad a_F' = d_w A_w, \quad a_B' = d_w A_w, \\ b_p &= -(u_e^* - u_w^*)\Delta y \Delta z - (v_n^* - v_s^*)\Delta x \Delta z - (w_f^* - w_b^*)\Delta x \Delta y \end{aligned} \tag{4.38}$$

De esta manera se tiene una ecuación para la corrección a la presión, con la que se completa el sistema de ecuaciones. La definición de b_p es exactamente la forma discreta de la ecuación de continuidad para u^*, v^* y w^*, que produce el método de volumen finito. Cuando este coeficiente es igual a cero, significa que dichas velocidades cumplen con la ecuación de continuidad. Este es el criterio de convergencia que usaremos durante todos los cálculos.

En problemas de convección natural la diferencia de temperaturas es la que promueve el movimiento, por lo tanto es importante resolver primero la ecuación de energía y luego las ecuaciones de cantidad de movimiento y la de corrección a la presión. Entonces, los pasos que sigue el método SIMPLEC, adaptado a problemas de convección natural son:

1. Se inicia con campos aproximados: T^*, p^*, u^*, v^* y w^*
2. Se resuelve la ecuación de energía para obtener T.
3. Se resuelven las ecuaciones de cantidad de movimiento usando los campos de presión y velocidad iniciales aproximados (p^*, u^*, v^*, w^*) y el campo de temperaturas T.
4. Se calculan los coeficientes de la ecuación de presión p_{nb} usando los coeficientes de las ecuaciones de cantidad de movimiento.
5. Se resuelve la ecuación de corrección a la presión.
6. Se corrige la presión mediante $p = p^* + p'$.
7. Se corrige la velocidad mediante ecuaciones (4.33), (4.35) y (4.36).

8. Se verifica el criterio de convergencia:

   a) Si $b_p \leq \epsilon_s$ ir al paso 9.

   b) Si $b_p > \epsilon_s$ regresar al paso 2

   donde ϵ_s es la tolerancia especificada.

9. Fin

Para problemas dependientes del tiempo se siguen los pasos mostrados en el diagrama de la figura 4.8.

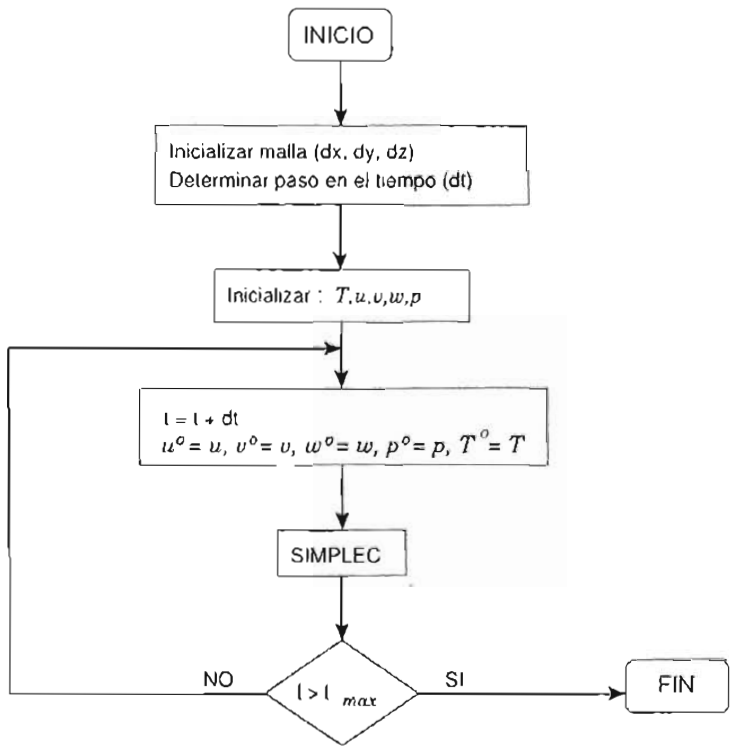


Figura 4.8: Diagrama de flujo de un problema dependiente del tiempo.

4.5. Solución de los sistemas lineales

La discretización de las ecuaciones gobernantes produce sistemas lineales como el mostrado en la ecuación (4.6). La complejidad y tamaño del conjunto de ecuaciones depende de la dimensionalidad del problema, el número de puntos de la malla y la estrategia de discretización. Aunque es posible utilizar cualquier procedimiento válido para resolver el conjunto de ecuaciones algebráicas, los recursos de cómputo disponibles son

una restricción muy fuerte. Existen dos familias de métodos para resolver los sistemas: directos e iterativos. Los métodos iterativos son generalmente mucho más económicos que los directos y por ello son preferidos cuando la matriz es dispersa. En este trabajo se ha usado un método iterativo que se basa en el algoritmo directo de Thomas o TDMA para matrices tridiagonales.

4.5.1. TDMA

El TDMA es un método directo para resolver de manera simple y eficiente sistemas tridiagonales. En una dimensión, la matriz del sistema es típicamente tridiagonal, por lo tanto el TDMA es aplicado directamente, véase figura 4.9.

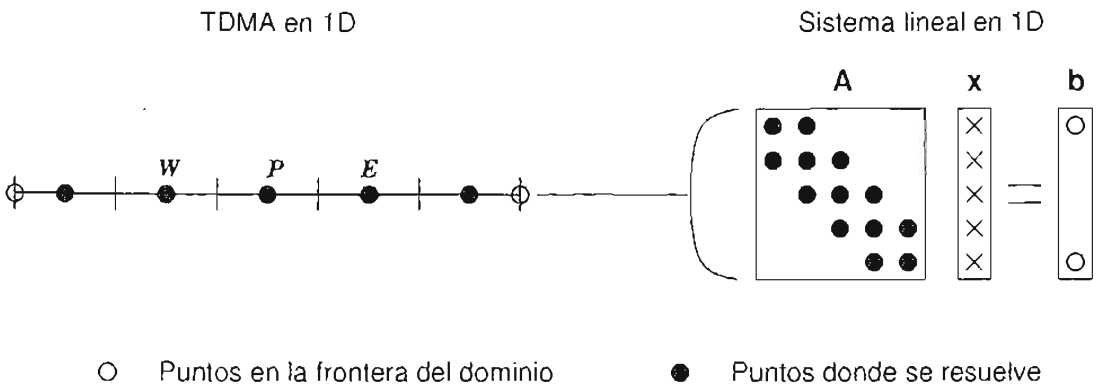


Figura 4.9: TDMA en 1D.

El TDMA puede resumirse como sigue: Considérese el siguiente sistema tridiagonal de $N \times N$:

$$\begin{pmatrix} a_1 & b_1 & 0 & 0 & 0 & \dots \\ c_2 & a_2 & b_2 & 0 & 0 & \dots \\ 0 & c_3 & a_3 & b_3 & 0 & \dots \\ 0 & 0 & c_4 & a_4 & b_4 & \dots \\ 0 & 0 & 0 & c_5 & a_5 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ \vdots \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ \vdots \end{pmatrix}$$

Para encontrar la solución del sistema anterior se realizan los siguientes pasos:

1. Calcular $P_1 = b_1/a_1$ y $Q_1 = d_1/a_1$
2. Desde i = 2 hasta i = N, calcular lo siguiente:

$$P_i = \frac{b_i}{a_i - c_i P_{i-1}}, \quad Q_i = \frac{d_i + c_i Q_{i-1}}{a_i - c_i P_{i-1}}. \tag{4.39}$$

3. Hacer $x_N = Q_N$

4. Desde i = N-1 hasta i = 1, hacer la sustitución hacia atrás:

$$x_i = P_i x_{i+1} + Q_i \tag{4.40}$$

4.5.2. Aplicación del TDMA en 2 y 3 dimensiones

El TDMA puede ser aplicado iterativamente, línea por línea, para resolver problemas en 2 y 3 dimensiones y es ampliamente usado en problemas de CFD. Considérese la figura 4.10 y la ecuación discretizada que tiene la forma:

$$a_P \phi_P = a_E \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + S_P. \tag{4.41}$$

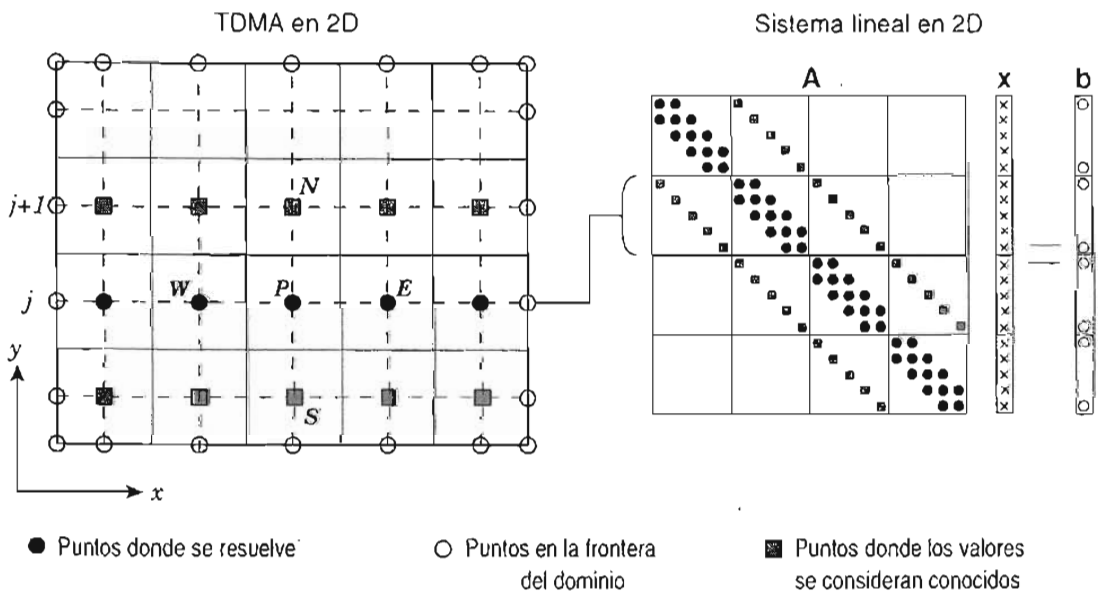


Figura 4.10: TDMA en 2D.

Para resolver el sistema, el TDMA es aplicado a lo largo de líneas horizontales o verticales. Por ejemplo, en la dirección x la ecuación (4.41) se rearregla de la siguiente forma:

$$-a_W \phi_W + a_P \phi_P - a_E \phi_E = a_N \phi_N + a_S \phi_S + S_P. \tag{4.42}$$

El lado derecho de esta última ecuación se supone conocido, y los valores de ϕ_N y ϕ_S se toman de la iteración anterior. La ecuación (4.42) representa un sistema tridiagonal donde $b = a_E$, $c = a_W$, $a = a_P$ y $d = a_N \phi_N + a_S \phi_S + S_P$. De esta manera el sistema puede resolverse a lo largo de la dirección x. Una vez resuelto el sistema en la línea

j, se actualizan los valores de ϕ y se resuelve la línea siguiente $j+1$. La secuencia en que las líneas se van resolviendo se conoce como la dirección de barrido, en este caso, dicha dirección es $+x$. El mismo procedimiento se realiza en la dirección y. El cálculo es repetido varias veces hasta obtener convergencia.

Para problemas tridimensionales el método se aplica línea por línea sobre un plano determinado y luego pasamos a un plano paralelo y continuamos el cálculo. Por ejemplo, para resolver a lo largo de la dirección x en un plano xz, véase figura 4.11, la ecuación discretizada se escribe como sigue:

$$-a_W\phi_W + a_P\phi_P - a_E\phi_E = a_N\phi_N + a_S\phi_S + a_F\phi_F + a_B\phi_B + S_P. \tag{4.43}$$

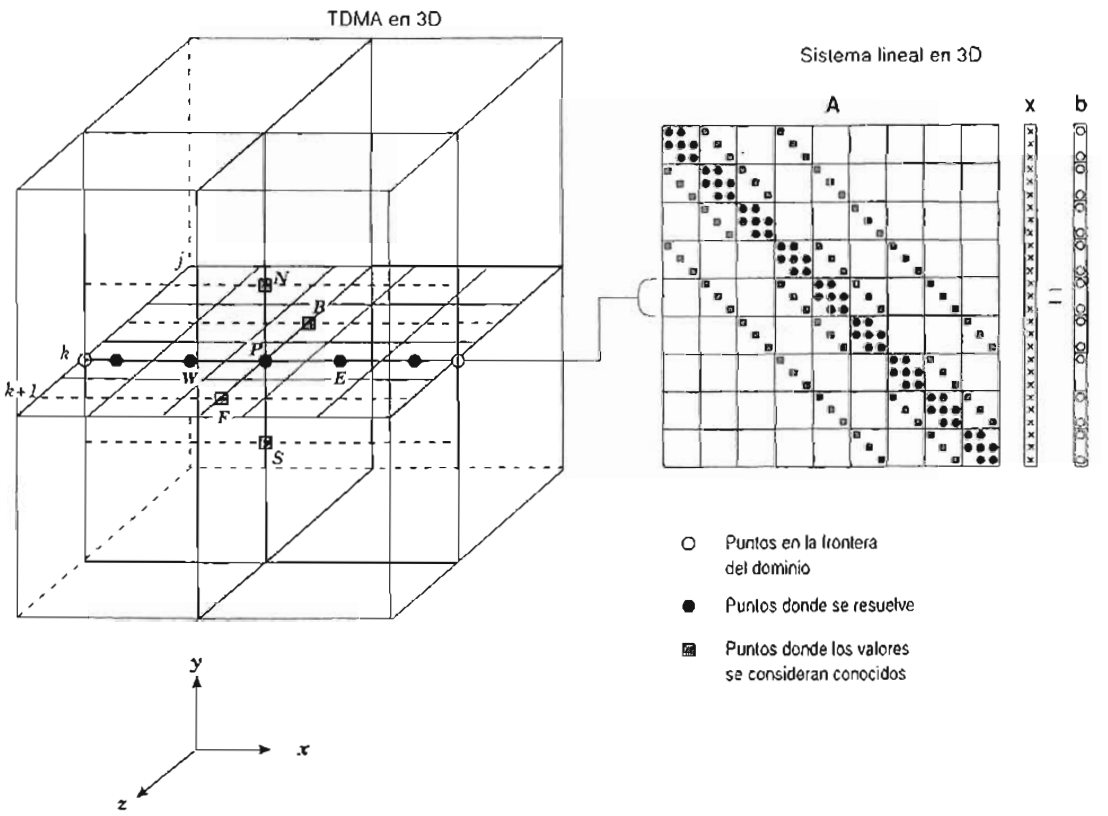


Figura 4.11: TDMA en 3D.

Los valores en N, S, F y B, del lado derecho, se consideran conocidos y son tomados de la iteración anterior. Así, los valores ϕ se calculan a lo largo de la dirección x mediante el TDMA. Luego, el cálculo se mueve de la línea k a la $k+1$ hasta cubrir todo el plano j. Después, se mueve el cálculo al plano $j+1$ hasta cubrir todo el dominio.

En dos y tres dimensiones la convergencia puede ser acelerada alternando la dirección de barrido de tal manera que toda la información de las condiciones de frontera se

transporte efectivamente dentro del dominio de estudio.

4.6. Discusión

En este capítulo hemos revisado la discretización de las ecuaciones de balance que se obtiene mediante la aplicación del método de volumen finito. La formulación discreta se hizo en términos de la ecuación general (3.1) y se revisaron diferentes esquemas numéricos para aproximar los términos difusivos y convectivos. La dependencia del tiempo se resolvió mediante un esquema completamente implícito (Backward Euler). La implementación de este tipo de esquemas es complicada e implica la solución de sistemas de ecuaciones, pero por otro lado, es incondicionalmente estable para cualquier tamaño del paso en el tiempo. Sin embargo, en algunos casos se requieren de pasos en el tiempo pequeños debido a que la precisión de este esquema es de primer orden. En general se recomienda el uso de esquemas implícitos para resolver problemas de flujos incompresibles dependientes del tiempo. Los esquemas explícitos (Forward Euler) en cambio, son mucho más simples de implementar, pero son condicionalmente estables de tal manera que, cuando la malla espacial es refinada, el paso en el tiempo se debe reducir para mantener la restricción que se impone sobre el número de Courant del esquema particular [28, 29]. Otros esquemas, como el de Crank-Nicholson y el de Adams-Bashforth utilizan aproximaciones de segundo orden y por lo tanto son más precisas, aunque también son condicionalmente estables.

La discretización de la parte difusiva de la ecuación general se hizo mediante diferencias centrales que proveen una aproximación de segundo orden. En los problemas que analizaremos la difusión es dominada por la convección, y por lo tanto no es necesario incorporar esquemas de más alto orden. Además, dada la forma de la implementación, que se explica en el capítulo siguiente, la inclusión de otros esquemas es relativamente simple. Los términos convectivos de la ecuación general deben de tratarse con mayor cuidado, debido a que representan la parte no lineal de las ecuaciones y es el efecto dominante en los problemas que estudiaremos. Para estos términos, se describieron tres esquemas: upwind (lineal), diferencias centrales (segundo orden) y QUICK (tercer orden). La idea es describir una implementación simple y fácil de modificar, en la que sea posible intercambiar los esquemas sin mucho trabajo. En el futuro se pueden incluir otros esquemas más precisos y estables como los que se describen en [25, 32, 26, 46, 60].

El algoritmo SIMPLEC para resolver el sistema de ecuaciones diferenciales parciales descrito aquí, es relativamente fácil de implementar y es usado con éxito en numerosos problemas de CFD. Otras variaciones, como el SIMPLER, pueden ahorrar trabajo computacional y su implementación es también simple. Sin embargo, el rendimiento de cada algoritmo depende de las condiciones del flujo y no es fácil determinar cual de ellos es mejor. Una comparación exahustiva de éstos y otros algoritmos para una variedad de flujos se puede encontrar en [39]. En general, se encuentra que el SIMPLER y el SIMPLEC muestran una convergencia muy robusta en problemas fuertemente acoplados, que es el caso de los problemas que nos interesan. La implementación de este trabajo intenta ser

clara y transparente, de tal manera que sea posible codificar, modificar, generalizar y sustituir cualquiera de estos algoritmos fácilmente.

Finalmente, aunque existen diversos algoritmos para resolver el sistema lineal de ecuaciones resultante de la discretización, aquí sólo se ha descrito uno muy sencillo. El TDMA es una generalización directa del algoritmo directo de Thomas para sistemas tridiagonales. La ventaja de este algoritmo es su simplicidad y además permite una paralelización directa en sistemas de memoria compartida. Esta característica es importante dado que, como se verá más adelante, el tiempo de cálculo que toma la solución del sistema llega a ser hasta 90 % del tiempo de cálculo total.

Capítulo 5

Proceso de desarrollo del sistema

En las últimas décadas el reuso de software científico ha sido una práctica muy común para realizar simulaciones numéricas en corto tiempo. Bibliotecas comerciales y de dominio público como BLAS (*Basic Linear Algebra Subroutines*) [5], LAPACK (*Linear Algebra Package*) [44], IMSL (*IMSL Mathematical and Statistical Libraries*) [34] y NAG (*NAG : Numerical Algorithms Group*) [55] son actualmente estándares en el desarrollo de códigos científicos. Estas bibliotecas están construidas en el estilo de programación estructurada usando lenguajes como C y Fortran con los que se obtiene un buen rendimiento. A pesar de la utilidad de este tipo de bibliotecas, se tienen algunos inconvenientes: 1) poca expresividad del código, es decir, la implementación no refleja claramente el algoritmo que se está usando, 2) repetición de código, esto es, en muchos casos se tiene que escribir el mismo código para tipos de datos distintos y sobre todo 3) alto acoplamiento y baja cohesión, es decir, las construcciones del código no reflejan claramente las matemáticas que implementa y se tienen muchas dependencias con otras partes del código. Estos tres problemas generan lo que se conoce como la *crisis del software*, [7], que es la falta de capacidad de crear software simple de entender, mantener y extender, y que además provea soluciones a problemas complejos.

En la actualidad existen diferentes estrategias para desarrollar software de una manera ordenada e incremental, que capturan desde el principio y durante el desarrollo de un sistema, los requerimientos deseados por los posibles usuarios. El Proceso Unificado [38], que incorpora las ideas de Booch [7], Jacobson[37] y Rumbaugh [63], se ha convertido en un estándar de desarrollo de software y se utiliza en el diseño de sistemas grandes. Este proceso provee una guía para la construcción eficiente de sistemas de calidad y consiste de 4 fases principales: Incepción, Elaboración, Construcción y Transición, las cuales se descomponen en proyectos pequeños que se van desarrollando en iteraciones. Estas iteraciones consisten de: 1) Requerimientos, 2) Análisis, 3) Diseño, 4) Implementación y 5) Pruebas, de tal manera que en cada iteración se entrega una versión del software que resuelve parte del problema, véase figura 5.1. Durante estas fases e iteraciones se obtiene una arquitectura, que contiene los aspectos dinámicos y estáticos más significativos del sistema. La arquitectura facilita la construcción de diferentes artefactos y componentes del sistema, y depende de muchos factores (plataforma de cómputo, sistema operativo,

componentes existentes, etc.). La arquitectura finalmente es una vista global del diseño del sistema que muestra las características más importantes, dejando de lado los detalles de la implementación.

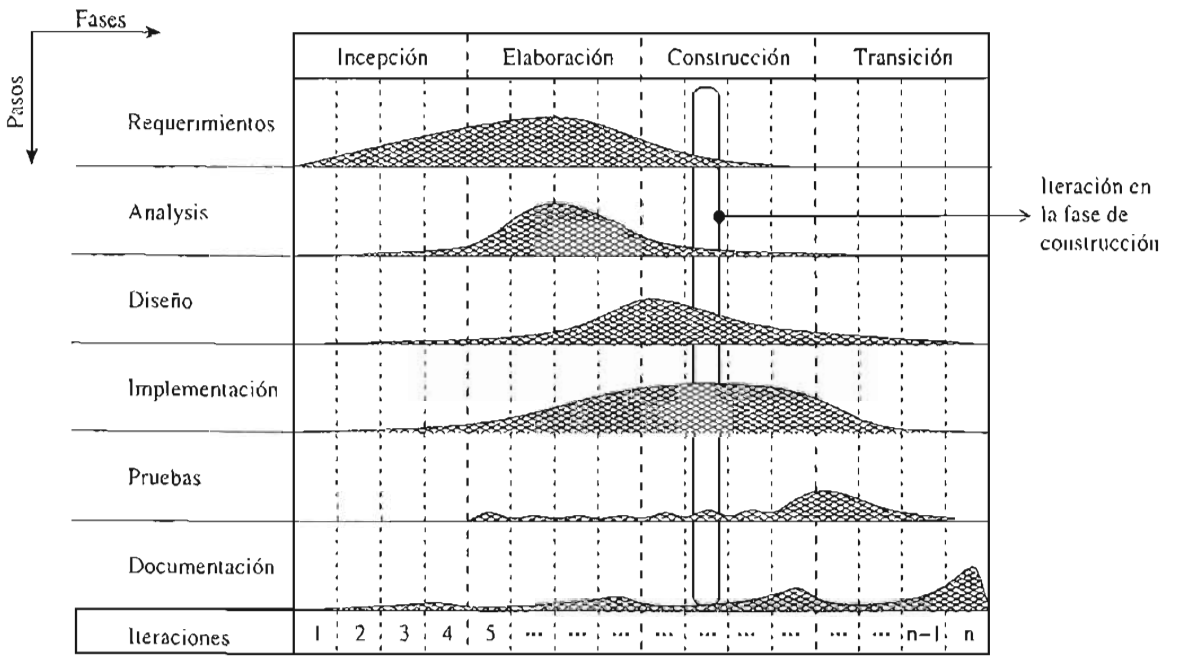


Figura 5.1: Etapas y pasos en el proceso unificado.

El Proceso Unificado provee una solución a la crisis del software, especialmente cuando se aplica en combinación con la Programación Orientada a Objetos. Mediante esta unión, es posible obtener sistemas de alta calidad, es decir, sistemas que son:

1. **útiles y de fácil uso** para encontrar soluciones a los problemas planteados,
2. **confiables**, es decir, contiene pocos errores (*bugs*),
3. **flexibles**, esto es que sea fácil y económico realizar cambios durante el desarrollo del sistema o después de la primera versión (durante el mantenimiento),
4. **accesibles económicamente** para obtenerlo y mantenerlo, y
5. **disponibles** en los diferentes sistemas de cada usuario.

Por otro lado, en el Proceso Unificado, la arquitectura del sistema se va definiendo durante el estudio de los casos de uso. En problemas de simulación numérica no es sencillo, y posiblemente no sea útil, determinar con facilidad los casos de uso, dado que una simulación requiere de altos recursos de cómputo y es casi imposible interactuar

mientras ésta se ejecuta. Se han ideado otras estrategias para atacar problemas de simulación numérica, que son modificaciones del Proceso Unificado. Por ejemplo, Schimmel [65] modifica el proceso de desarrollo definido por Larman [45], que está basado en el proceso unificado, para incluir explícitamente el modelo matemático y la discretización como actividades de desarrollo. La figura 5.2 describe el proceso de desarrollo propuesto por Shimmel. Cuando el desarrollo de software requiere de resultados en poco tiempo, y sólo se tiene a un equipo reducido de personas, se recomienda el uso del modelo de la programación extrema [85, 86, 4] (XP por *eXtreme Programming*), en donde pequeñas partes del sistema se van implementando y modificando constantemente hasta que cumplen con los requerimientos del usuario. Desarrollos cortos y ciclos de pruebas constantes son inherentes a la XP, de tal manera que el sistema crece a partir de la experiencia. Muchos de los problemas surgen sólo cuando se implementa, se compila y se ejecuta una parte del código y no durante la etapa de análisis.

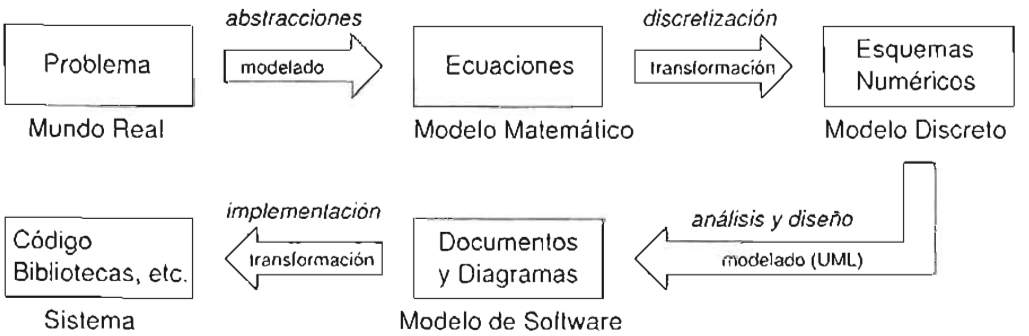


Figura 5.2: Modelo de desarrollo donde se incluyen el modelo matemático y la discretización [65].

5.1. Proceso de desarrollo en aplicaciones científicas

El objetivo principal de este trabajo es elaborar un sistema orientado a objetos, genérico y paralelo, para resolver las ecuaciones de balance descritas en el capítulo 3. Este objetivo se logra mediante la combinación del modelo de Schimmel con el de la programación extrema para obtener resultados en tiempos cortos. El modelo de desarrollo de software que hemos concebido en este trabajo, se puede aplicar a problemas de simulación numérica en general, dado que incluye, al igual que el de Schimmel, los modelos matématico y discreto, y considera además la generalización y la optimización como actividades de desarrollo, véase figura 5.3.

En la figura 5.3 se muestran las actividades que hemos seguido para desarrollar nuestro sistema. Las etapas principales de este proceso son:

Descripción del problema : es importante siempre describir en forma clara el problema a resolver. Aquí se plantean los objetivos y metas que se desean alcanzar durante el desarrollo del sistema.

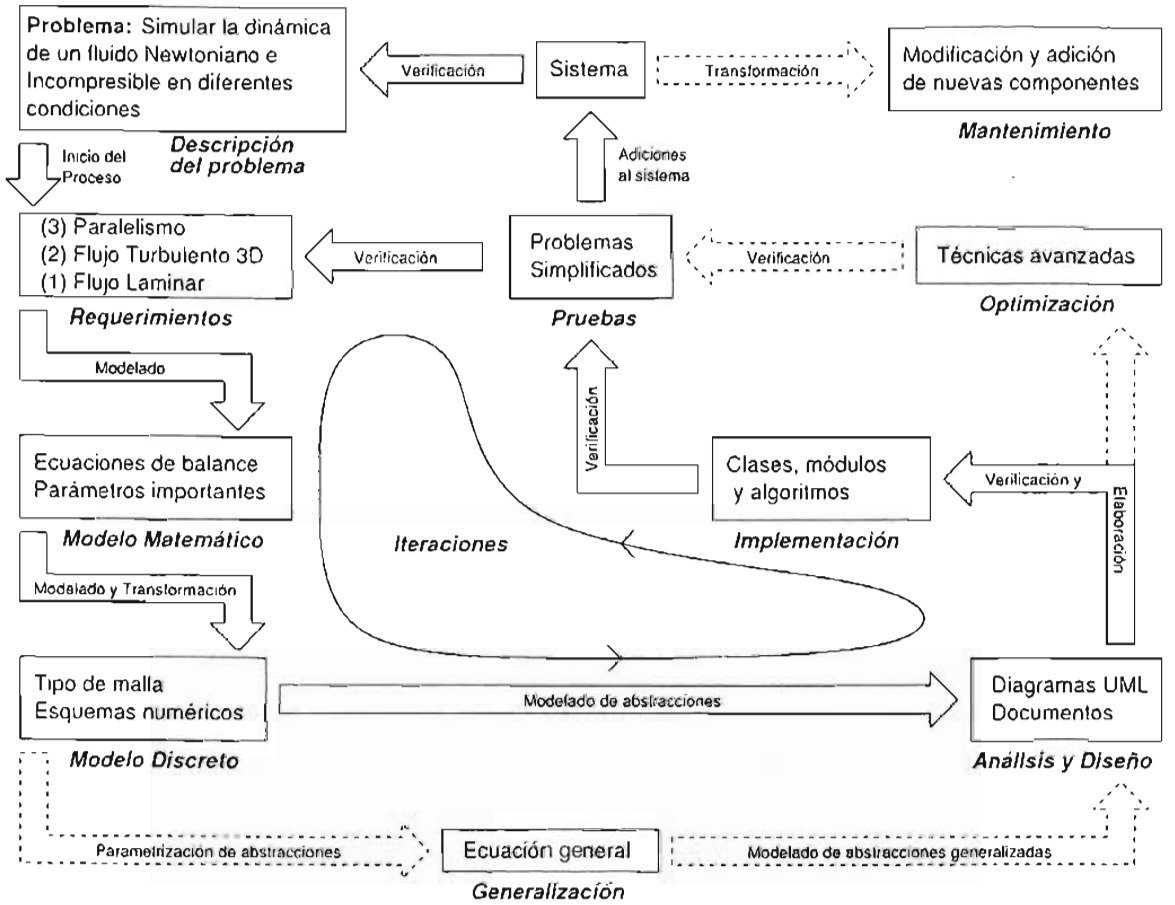


Figura 5.3: Proceso de desarrollo ideado para construir el sistema de este trabajo.

Requerimientos : aquí se especifican en detalle cada uno de los problemas que deseamos resolver. Esto es un simil de los casos de uso utilizados en el desarrollo de sistemas de tiempo real. En este trabajo, los "casos de uso" son problemas particulares, los cuales se complican conforme se avanze en la construcción del sistema.

Modelo matemático : cada problema tiene un conjunto de ecuaciones matemáticas particular, las cuales se describen en esta etapa y se definen algunos de sus parámetros. Estos parámetros se utilizarán posteriormente en la etapa de generalización.

Modelo discreto : en esta etapa se discretiza el espacio geométrico donde se va a resolver el problema y a partir de esta discretización, se obtienen ecuaciones discretas aplicando algún método (Volumen Finito, Diferencias Finitas, Elemento Finito, etc.) y diferentes esquemas numéricos. También se plantean los métodos de solu-

ción de las ecuaciones discretas.

Análisis y Diseño : una vez que se ha detallado el problema a resolver se hace un análisis y un diseño orientado a objetos, generando información para documentar el problema (tarjetas CRC, diagramas UML, bitácoras, etc.).

Implementación : haciendo uso de la documentación obtenida en el paso anterior, se comienza la implementación de los elementos (clases, funciones, espacios de nombres, etc.) requeridos para resolver el problema en cuestión.

Pruebas : con la primera versión de los elementos construidos en la etapa anterior, se realizan pruebas resolviendo problemas simplificados. La solución de estos problemas se verifican con los requerimientos descritos en la primera etapa. Cada vez que sea posible se compara con resultados conocidos o con resultados experimentales. Las pruebas se realizan en cada iteración tratando de encontrar posibles fallas cometidas en las etapas anteriores.

En nuestro desarrollo, se estudia un problema sencillo a la vez y se realizan varias iteraciones hasta obtener un producto lo suficientemente estable. Estos problemas simplificados se pueden ver como casos de uso del sistema. La solución de cada uno de estos problemas produce versiones preliminares y cumple sólo algunos de los requerimientos. Después se consideran problemas más complejos y se repite el proceso. En este proceso las etapas de implementación y pruebas son muy importantes y es en donde se pasa la mayor parte del tiempo. En sistemas de mayor magnitud, en donde se tiene un equipo de almenos cuatro personas, se recomienda hacer estas dos etapas por "pares" (*pair programming*) como en la programación extrema, y hacer intercambio de miembros de pares en tiempos cortos.

Cuando se han estudiado varios problemas (en 1D, 2D y 3D) y se tiene una versión estable del sistema se pueden incorporar las siguientes etapas:

Generalización : a partir de la solución de problemas independientes, se realizar un análisis de los parámetros importantes definidos en cada uno y se intenta hacer una generalización de estos. Esto produce ecuaciones generales y nos lleva a la reutilización de software, que es una de las características que deseamos en nuestro sistema.

Optimización : una vez que se han revisado todos los problemas que nos interesan, se agrega al proceso la actividad de optimización. Después de la optimización, se obtiene la primera versión completa del sistema y posteriormente es posible modificar o incluir nuevas componentes al sistema de acuerdo con las necesidades de los usuarios. Este paso es muy importante, pues se requiere que el código ejecutable sea eficiente. Aquí se utilizan algunas de las técnicas descritas en la sección 2.3.

En la última etapa del desarrollo, que es la de mantenimiento y puede iniciar un nuevo proceso de desarrollo que comienza con la descripción de los nuevos problemas a resolver y la especificación de nuevos requerimientos.

5.2. Arquitectura del sistema

El proceso unificado toma en cuenta los riesgos posibles que se tienen en el desarrollo de software, y permite realizar iteraciones como un medio para controlar ese riesgo. Además, se basa en el diseño y descripción de arquitecturas sobre las cuales se elaboran las diferentes componentes [1] del sistema. Es importante entonces, muy al inicio del desarrollo, diseñar y describir una arquitectura central a partir de la cual se determinen las componentes o módulos que deben ser desarrollados. Los módulos deben tener baja dependencia entre ellos para permitir un desarrollo y mantenimiento simple. Un sistema con muchas dependencias tiene un alto acoplamiento que lo hace difícil de elaborar y mantener. Un buen sistema tiene entonces bajo acoplamiento y los cambios en un módulo no se propagan fácilmente a otros módulos. La encapsulación permite el bajo acoplamiento entre los módulos debido a que la mayor parte de la implementación está escondida y sólo se puede interacturar mediante una interfaz bien definida que no cambia. Un módulo bien diseñado tiene la propiedad de que sus interfaces proveen una abstracción de alguna entidad bien entendida, pero que sin embargo, su implementación puede ser muy compleja. Los módulos que cumplen con esta propiedad se dice que tienen alta cohesión.

La arquitectura de un sistema es la estructura global que determina la manera en que se desarrollarán las diferentes componentes del software. Un desarrollo basado en componentes permite que éstas puedan ser reutilizadas, modificadas o sustituidas en el futuro sin afectar otras partes. Por supuesto que la reusabilidad de una componente depende también de factores técnicos y del contexto donde fue desarrollada. El contexto lo determina la arquitectura del sistema. Por ejemplo, si dos componentes son desarrolladas usando una misma arquitectura, éstas podrán ser intercambiadas fácilmente, mientras que si se desarrollan con arquitecturas diferentes posiblemente no podrán intercambiarse.

La arquitectura central del sistema de este trabajo se diseñó a partir de un esquema general de modelación computacional numérica para problemas de física. El diagrama de la figura 5.4 muestra esquemáticamente la forma en que se lleva a cabo una modelación numérica en general.

Se observa, en la figura 5.4, que existe una división natural de las componentes principales que deben existir en un sistema de este tipo. En este caso la simulación numérica se separa en tres componentes principales:

Modelo matemático: Describe mediante un conjunto de ecuaciones el comportamiento del fenómeno bajo estudio. En CFD, se tiene generalmente un sistema de ecuaciones diferenciales parciales no lineales y acopladas. En el caso de fluidos newtonianos e incompresibles, las ecuaciones pueden generalizarse y escribirse como en la ecuación (3.1). A partir de la ecuación general, se pueden obtener las ecuaciones

[1] Existen diferentes definiciones de componente, aquí se pensará una componente como una entidad que se puede reusar o reemplazar.

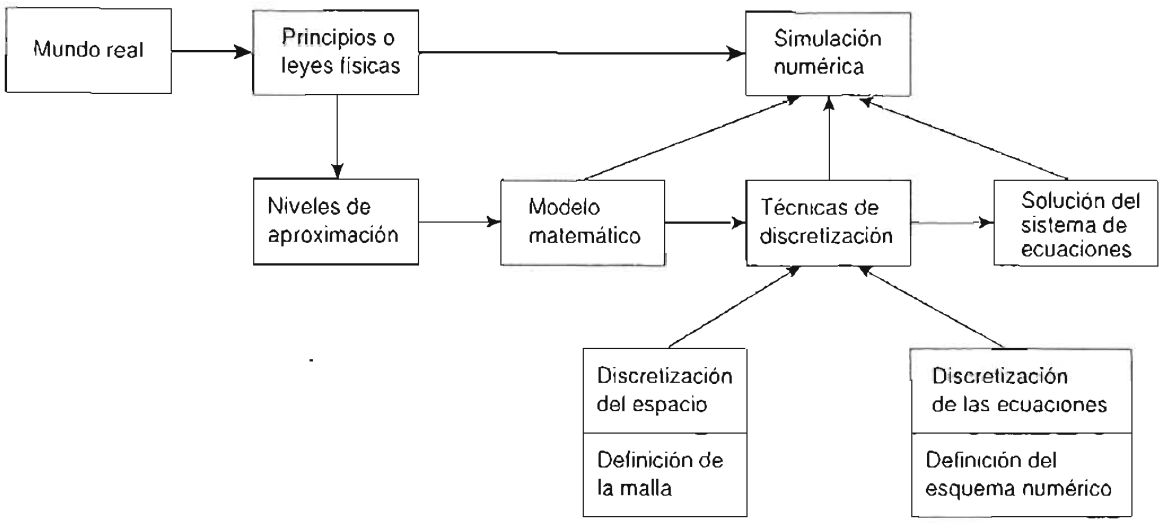


Figura 5.4: Diagrama general de una modelación computacional [28].

que intervienen en la dinámica de un flujo mediante la sustitución de algunos parámetros, como se explicó en el capítulo 3.

Técnicas de discretización: En una simulación numérica se debe decidir como aproximar las soluciones de las ecuaciones que definen al problema. Esta decisión depende de la forma en que se haga la discretización del dominio de estudio (malla del dominio). Existen básicamente dos tipos de mallas: estructuradas y no estructuradas. Dada la malla, es posible elegir entre varios esquemas numéricos, cada uno de ellos con propiedades convenientes para problemas particulares. Es importante entonces, poder intercambiar fácilmente los esquemas numéricos para encontrar el más adecuado a cada problema.

Solución del sistema de ecuaciones: Después de la discretización, casi todos los problemas se transforman en un sistema de ecuaciones algebraicas. Incluso, problemas de otras áreas se reducen a resolver sistemas de ecuaciones similares. Actualmente, se han desarrollado múltiples bibliotecas que incluyen implementaciones de algoritmos sofisticados para resolver este tipo de sistemas. Dada la generalidad, es importante que los algoritmos puedan actuar sobre diferentes tipos de datos sin mayor problema, por lo que el desarrollo de algoritmos genéricos facilita el acoplamiento con otros programas.

La arquitectura del sistema de este trabajo se definió tomando como base el esquema de la figura 5.4 y las descripciones anteriores de las tres componentes principales de una simulación numérica. Un diagrama de la arquitectura del sistema se muestra en la figura 5.5, donde se pueden ver tres componentes principales que corresponden con

aquellas mostradas en el diagrama de la figura 5.4. En la componente que se refiere al modelo matemático, se define una ecuación general que encapsula las características importantes y comunes de las ecuaciones [2]. Es natural, entonces reutilizar esta ecuación general, para describir las ecuaciones particulares de cada problema. En una simulación se trabaja con la versión discreta de estas escuaciones y su forma depende de la técnica de discretización que se utilice.

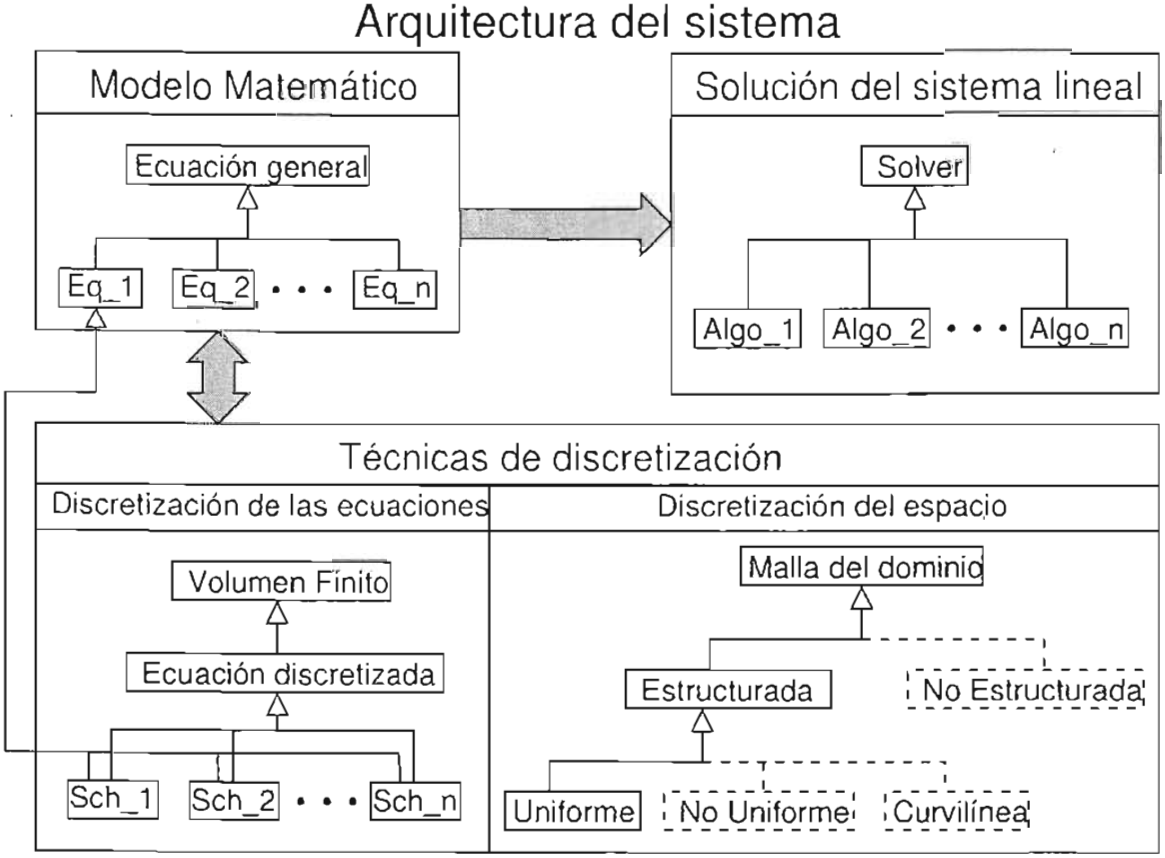


Figura 5.5: Diagrama de la arquitectura sobre la que se desarrolló el sistema.

La componente principal de las técnicas de discretización, se descompone en dos partes: discretización de las variables dependientes (espacio) y discretización de las variables independientes (ecuaciones). La primera corresponde a la creación de la malla del dominio de estudio. En este trabajo sólo se tratan las mallas estructuradas uniformes, sin embargo agregar otro tipo de mallas no es complicado, sólo es necesario definirlas correctamente dentro de la arquitectura del sistema (líneas punteadas en la figura 5.5).

[2]En este trabajo sólamente se toman en cuenta las características de las ecuaciones que describen a fluidos newtonianos e incompresibles.

La información de la malla se utiliza para definir los esquemas numéricos con que se discretizarán los diferentes términos de las ecuaciones. En este caso se utilizó el método de volumen finito y diferentes esquemas para aproximar los términos convectivos y difusivos.

Los esquemas numéricos de discretización dependen de las características particulares de las ecuaciones que se estén trabajando. Estas características se definen tanto en la ecuación general como en cada una de las ecuaciones particulares al problema. Por otro lado, los coeficientes de las ecuaciones discretas dependen del esquema numérico, por lo que existe una dependencia circular entre los esquemas numéricos y las ecuaciones discretas. La manera en que se resuelve esta dependencia es mediante la definición de ecuaciones discretas que se parametrizan con el esquema numérico. Así es posible adaptar las ecuaciones con el esquema numérico que se desee. Los esquemas numéricos funcionan entonces como adaptadores de las ecuaciones discretas; por ejemplo, en las definiciones `Eq_1<Sch_1>` y `Eq_1<Sch_2>` se define la misma ecuación `Eq_1<>`, adaptada con los esquemas `Sch_1` y `Sch_2` respectivamente. Cada definición se comporta de manera diferente a la hora de calcular los coeficientes. Tenemos entonces un comportamiento polimórfico de las clases que definen las ecuaciones. Este es un polimorfismo de tipo estático, de tal manera que no hay fuentes de bajo rendimiento asociadas con este comportamiento. Está técnica de polimorfismo estático es muy usada para optimizar código, véase por ejemplo [81].

Una vez hecha la discretización, se obtiene un sistema algebraico de ecuaciones que se resuelve usando un algoritmo conveniente. Muchos algoritmos genéricos han sido implementados, en diferentes lenguajes, y es posible utilizarlos en este desarrollo, siempre y cuando se tengan los argumentos necesarios para hacerlos funcionar. En general, un sistema de ecuaciones se escribe como $A\mathrm{x} = \mathrm{b}$. Un algoritmo genérico, además de algunos parámetros para la convergencia, sólo necesita conocer las entradas de la matriz A y del vector b para calcular la solución x. El algoritmo genérico se debe encargar de manejar los diferentes tipos de números (`int`, `float`, `double`, `complex`) y la forma de las matrices (densa, banda, dispersa, triangular, etc.).

Un glosario de las clases y algoritmos implementados con la arquitectura antes descrita, así como los diagramas de clases en notación UML, se pueden consultar en el apéndice A.

5.3. Acoplamiento y no linealidad de las ecuaciones

En el capíutlo 3 se describió el algoritmo SIMPLEC que es utilizado para desacoplar las ecuaciones (acoplamiento presión – velocidad) y resolver la no linealidad de las mismas. Este algoritmo no está explícitamente incluido en el diseño del sistema. La razón es que la discretización, la construcción de la malla, los esquemas numéricos y la solución de los sistemas han sido encapsulados en diferentes clases y algoritmos genéricos, de tal manera que las abstracciones son de un nivel tal que es posible escribir el SIMPLEC o sus variantes sin ninguna distracción y obtener una implementación clara

y concisa. Como ejemplo véase el código de la sección 4.4.3 y compárese con el algoritmo descrito en la sección 6.2.

De igual manera se pueden implementar otros algoritmos, como el CLEAR [71], para desacoplar las ecuaciones y comparar los resultados de ambos.

5.4. Descomposición de dominio y paralelismo

El término descomposición de dominio (DD) es utilizado de forma diferente por especialistas en análisis numérico de ecuaciones diferenciales parciales. En general, DD es un nombre genérico que es usado para describir varios tipos de algoritmos. En este trabajo usaremos el término DD para referirnos al caso en el que el dominio espacial en donde se resuelve el problema, es dividido en un cierto número de bloques o subdominios los cuales pueden asignarse a procesadores independientes.

Existen varias motivaciones para utilizar DD:

- La paralelización es fácil y se puede obtener un buen escalamiento.
- Se pueden simplificar problemas en geometrías complejas.
- Se pueden utilizar diferentes modelos en diferentes subdominios.
- Se pueden refinar las mallas localmente en cada subdominio.
- Se reducen los requerimientos de hardware (memoria y cpu), pues cada subdominio es mucho más pequeño que el dominio global.

Se pueden encontrar diferentes clasficaciones para métodos de DD, véase por ejemplo [64]. Un criterio importante para la clasificación es de acuerdo a la etapa en donde se lleva a cabo la descomposición:

1. La descomposición se realiza antes de la discretización. Aquí se utilizan subdominios traslapados y se resuelve el mismo problema en cada uno de ellos, pero con condiciones de frontera adicionales en las interfaces. Estas condiciones de frontera deben ser actualizadas en cada iteración, de tal manera que en cada subdominio se obtiene una solución local, la cual converge a la solución global. Este enfoque es una versión numérica paralela del método alternante de Schwarz [11, 67].

2. La decomposición se hace después de la discretización. En este enfoque se considera un problema global sobre el dominio completo. Cada procesador es usado para generar las ecuaciones discretas relacionadas con su parte del dominio. Entonces, se utiliza un algoritmo paralelo para resolver las ecuaciones discretas, que en un principio se reducen a un sistema lineal de ecuaciones.

El primer enfoque tiene dos ventajas: *(i)* es un paralelismo de grano grueso e implica menos comunicaciones entre los procesadores (sólo después de cada iteración temporal) y

(ii) permite el reuso de la mayor parte de las clases desarrolladas para resolver problemas en un sólo procesador. Dadas estas ventajas, la paralelización que se utilizó en este trabajo se basa en el enfoque 1.

El sistema desarrollado aquí contiene un conjunto de clases para descomponer el dominio en varios subdominios y generar su propia discretización (malla y esquemas numéricos). Una vez que se ha realizado la descomposición, sobre cada subdominio se resuelve el mismo problema pero usando condiciones de frontera ficticias sobre las interfases de los subdominios. La figura 5.6 muestra esquemáticamente la manera en que se realiza la partición del dominio.

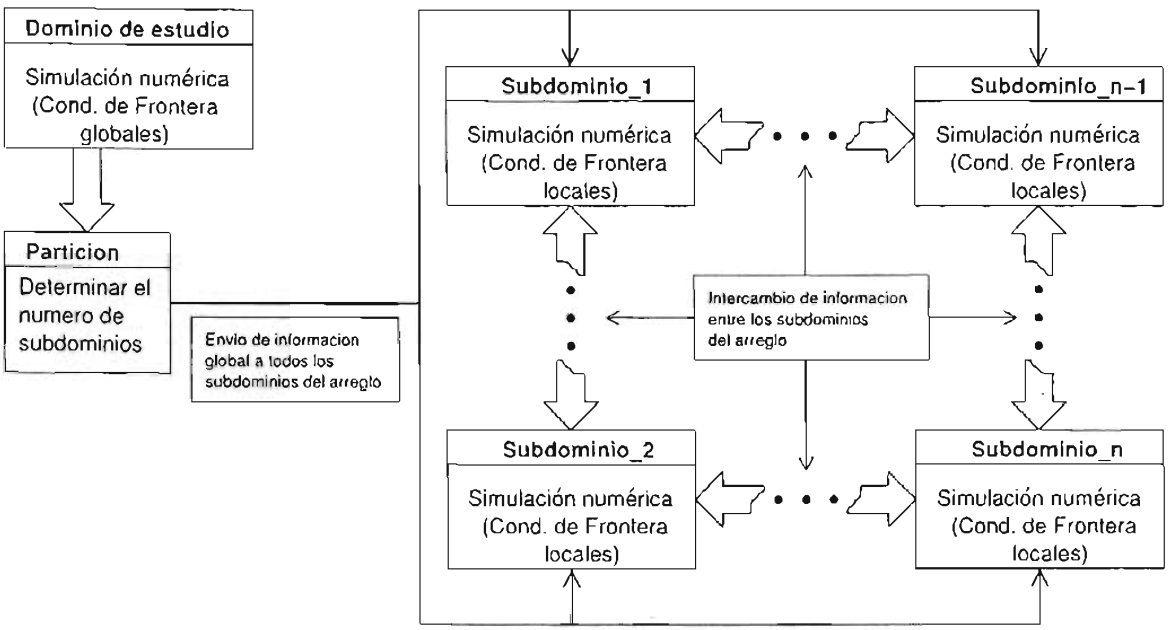


Figura 5.6: Partición del dominio.

Aún cuando la paralelización sea aparentemente simple, el desarrollo de algoritmos paralelos y su implementación, requiere del análisis detallado de la forma de la partición del dominio y de las comunicaciones que serán necesarias entre cada uno de los subdominios. Además, se requiere realizar un análisis de la formulación de los subproblemas para cada subdominio, de tal manera que se haga un planteamiento correcto en cada uno de ellos. Un planteamiento inadecuado llevará a soluciones no realistas.

5.4.1. Algoritmo alternante de Schwarz

El método de Schwarz fue introducido por primera vez en 1870, que aunque no es originalmente un método numérico, puede ser usado para resolver numéricamente ecuaciones diferenciales parciales.

Considere el dominio que se muestra en la figura 5.7, en donde se desea resolver el siguiente problema:

$$\begin{aligned} L\Phi &= F \text{ en } \Omega, \\ \Phi &= G \text{ sobre } \delta\Omega. \end{aligned} \tag{5.1}$$

donde L es un operador diferencial, Φ es una variable escalar, F es un término fuente y G es el valor de Φ en la frontera. En esta descripción nos restringimos a condiciones de frontera de tipo Dirichlet, aunque se pueden incluir otro tipo de condiciones fácilmente.

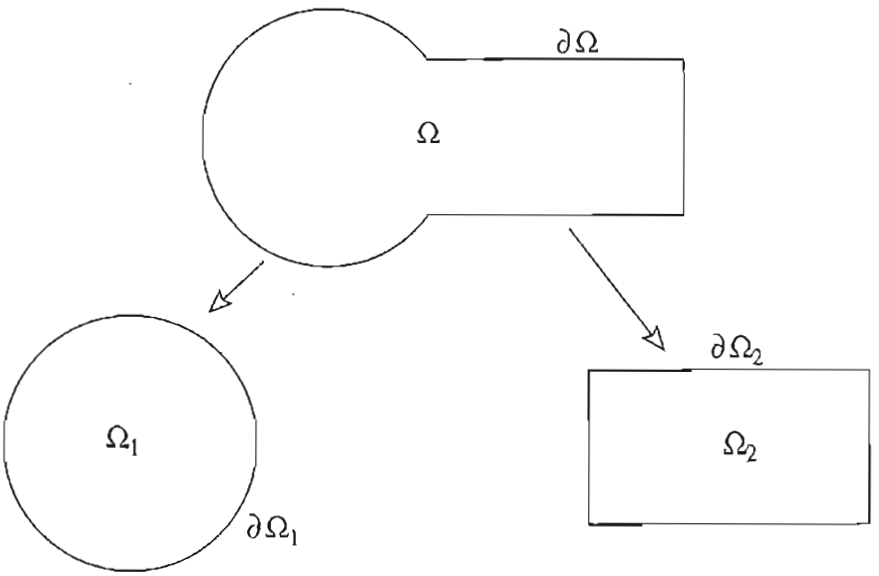


Figura 5.7: Dominio de estudio y su descomposición

El dominio se descompone en dos subdominios: $\Omega = \Omega_1 \cup \Omega_1$. La nomenclatura que se usará en lo que sigue es la siguiente, véase figura 5.8:

- Los dominios $\Omega, \Omega_1, \Omega_1$ no contienen su frontera.
- $\delta\Omega$ es la frontera real de Ω.
- $\bar{\Omega} = \Omega \cup \delta\Omega$ es la cerradura del dominio.
- Γ_i es una frontera artificial y es parte de la frontera de Ω_i que está en el interior de Ω.
- $\delta\Omega_i \backslash \Gamma_i$ es la frontera de Ω_i sin la parte de Γ_i.
- Φ_i^n denota la solución aproximada en $\bar{\Omega}_i$ después de n iteraciones.

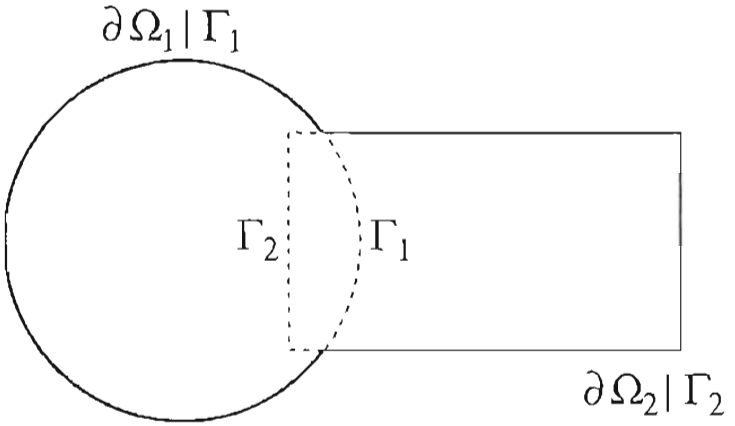


Figura 5.8: Dominios usados en el algoritmo alternante de Schwarz.

- $\Phi_i^n|_{\Gamma_j}$ es la restricción de Φ_i^n en Γ_j, donde $i \neq j$.

El método alternante de Schwarz comienza seleccionando una solución inicial en Ω_2, que llamamos Φ_2^0 (en realidad sólo se necesita conocer los valores de Φ_2 en Γ_1). Entonces, se resuelve iterativamente, para $n = 1, 2, 3, ...$, el problema

$$\begin{aligned} L\Phi_1^n &= F \text{ en } \Omega_1 \\ \Phi_1^n &= G \text{ sobre } \partial\Omega_1 \backslash \Gamma_1 \\ \Phi_1^n &= \Phi_2^{n-1}|_{\Gamma_1} \text{ sobre } \Gamma_1 \end{aligned}$$

para Φ_1^n, seguido por la solución del problema

$$\begin{aligned} L\Phi_2^n &= F \text{ en } \Omega_2 \\ \Phi_2^n &= G \text{ sobre } \partial\Omega_2 \backslash \Gamma_2 \\ \Phi_2^n &= \Phi_1^n|_{\Gamma_2} \text{ sobre } \Gamma_2 \end{aligned}$$

De esta manera, en la mitad de cada iteración del método alternante de Schwarz, se resuelve un problema en el subdominio Ω_i con la condición g en la frontera real $\partial\Omega_i \backslash \Gamma_i$, y valores de la iteración anterior en la frontera Γ_i.

Supongamos ahora que las ecuaciones han sido discretizadas usando algún método, volumen finito por ejemplo. Entonces el vector asociado con Φ_i tenemos el vector discreto de coeficientes:

$$\phi_i = \begin{pmatrix} \phi_{\Omega_i} \\ \phi_{\partial\Omega_i \backslash \Gamma_i} \\ \phi_{\Gamma_i} \end{pmatrix} \tag{5.2}$$

Los coeficientes $\phi_{\partial\Omega_i\setminus\Gamma_i}$ son conocidos, y están dados por los valores del dominio Ω_j con $i \neq j$. Los vectores discretos correspondientes a F y G son f_i y g_i respectivamente. La matriz de coeficientes A_i es la discretización del operador L en Ω_i. Esta matriz tiene tres componentes: $A = (A_{\Omega_i}, A_{\partial\Omega_i\setminus\Gamma_i}, A_{\Gamma_i})$. La forma discreta de $\phi_i^n|_{\Gamma_j}$ la escribimos como W_j^n. Entonces, en forma discreta, un algoritmo alternante de Schwarz se escribe como sigue:

Algoritmo alternante de Schwarz

$W_1^0 \leftarrow 0.$
For $n = 1, \ldots$
  Resolver para ϕ_1^n:
    $A_1\phi_1^n = f_1$ en Ω_1
    $\phi^n_{\partial\Omega_1\setminus\Gamma_1} = g_1$ sobre $\partial\Omega_1\setminus\Gamma_1$
    $\phi^n_{\Gamma_1} = W_1^{n-1}$ sobre Γ_1
  $W_2^n \leftarrow \Phi_1^n|_{\Gamma_2}$
  Resolver para ϕ_2^n:
    $A_2\phi_2^n = f_2$ en Ω_2
    $\phi^n_{\partial\Omega_2\setminus\Gamma_2} = g_2$ sobre $\partial\Omega_2\setminus\Gamma_2$
    $\phi^n_{\Gamma_2} = W_2^n$ sobre Γ_2
  $W_1^n \leftarrow \Phi_2^n|_{\Gamma_1}$
  Checar la convergencia:
    Si $||W_1^n - W_1^{n-1}|| \leq \text{tol}_{\Gamma_1}$ y $||W_2^n - W_2^{n-1}|| \leq \text{tol}_{\Gamma_2}$ Fin.
    Si $||\phi_1^n - \phi_1^{n-1}|| \leq \text{tol}_{\Omega_1}$ y $||\phi_2^n - \phi_2^{n-1}|| \leq \text{tol}_{\Omega_2}$ Fin.
End For

Nótese que este algoritmo es serial, en donde cada iteración se divide en dos partes: en la primera se resuleve el problema en el Ω_1, y en la segunda en el Ω_2. Después de obtener la solución en Ω_1 se actualiza W_2^n para poder resolver el siguiente problema en Ω_2. Una vez obtenida una solución en Ω_2 se checa la convergencia, que en este caso se revisa que tanto la diferencia de las ϕ's y de las W's en iteraciones subsecuentes sea menor que una tolerancia dada.

En este caso se supone que las mallas de los subdominios coinciden en el traslape. En el caso de que esto no suceda, se debe definir un operador de interpolación entre mallas, véase por ejemplo [67].

5.4.2. Algoritmo paralelo alternante de Schwarz

El algoritmo alternante de Schwarz descrito antes es serial, pero se puede construir fácilmente un algoritmo paralelo, en donde se pueden considerar más de dos subdominios. Considere por ejemplo la partición del dominio mostrada en la figura 5.9. En este caso se tienen cuatro subdominios en donde las mallas son coincidentes y por lo tanto no es necesario ningún operador de interpolación.

El algoritmo paralelo que utilizamos en este trabajo para K subdominios, es como sigue:

Algoritmo paralelo alternante de Schwarz

01 Particionar el dominio
02 Definir el problema en cada subdominio
03 $W_1^0 \leftarrow 0, ..., W_K^0 \leftarrow 0$
04 Parallel For $k = 1, ..., K$
05 For $n = 1, ...$
06 Resolver para ϕ_k^n:
07 $A_k \phi_k^n = f_k$ en Ω_k
08 $\phi^n_{\partial\Omega_k \backslash \Gamma_k} = g_k$ sobre $\partial\Omega_k \backslash \Gamma_k$
09 $\phi^n_{\Gamma_k} = W_k^{n-1}$ sobre Γ_k
10 Checar la convergencia :
11 Si $||W_k^n - W_k^{n-1}|| \leq \text{tol}_{\Gamma_k}$ Fin
12 Si $||\phi_k^n - \phi_k^{n-1}|| \leq \text{tol}_{\Omega_k}$ Fin
13 End For
14 Wait
15 Enviar $\phi_k^n|_{\Gamma_{k-1}}$ a los subdominios vecinos
16 $W_k^0 \leftarrow \phi_{nb}^n|_{\Gamma_k}$
17 End Parallel For

En el algoritmo anterior primero se particiona el dominio. Después se define el problema en cada subdominion. Posteriormente, se inicializan las condiciones de frontera ficticias en las interfases, en este caso todas igual a cero, línea 3. En la línea 4 se inicia la solución en paralelo de los problemas definidos en cada subdominio (que es el mismo problema con diferentes condiciones de frontera). El For interno termina cuando el criterio de convergencia local se satisface. Debido a que en general los subdominios no terminan en el mismo número de iteraciones internas, es necesario poner una "barrera" para esperar hasta que todos los subdominios hayan resuelto su problema, línea 14. Cuando todos los subdominios han resuelto su problema, se intercambia la información correspondiente entre los subdominios vecinos. Esta información la denotamos $\phi_k^n|_{\Gamma_{k-1}}$. En la figura 5.9 se observa la información que es necesario intercambiar. En la línea 16 se actualizan las condiciones de frontera para continuar con el proceso. Los problemas que resolvemos son dependientes del tiempo, por lo que es necesario adicionar un ciclo global (entre las líneas 03 y 04) que tome en cuenta las iteraciones temporales.

5.4.3. Paralelización dentro del sistema

La paralelización es una componente adicional del sistema desarrollado en este trabajo y contiene clases para definir el dominio global de estudio, los subdominios junto con las comunicaciones necesarias entre cada uno de ellos y los tipos de partición que se pueden realizar.

Para resolver un problema en paralelo, primero se genera el dominio global de estudio, después se determina el tipo de partición conveniente. La partición se aplica sobre el dominio global y se definen los parámetros para cada ecuación. Las ecuaciones se definen de manera similar que en el problema serial en cada uno de los subdominios, pero las condiciones de frontera deben tratarse de manera distinta en las interfases entre subdominios. La figura 5.9 muestra la forma en que se definen las fronteras en las interfases.

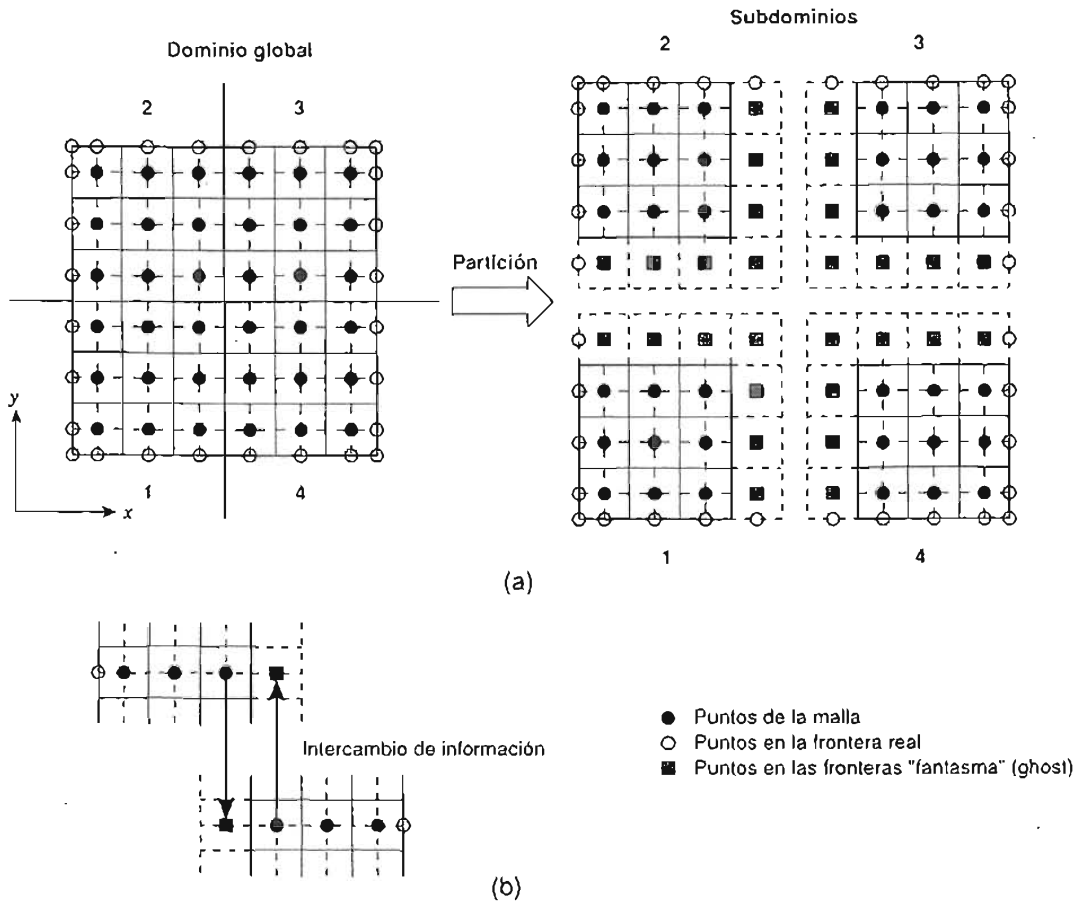


Figura 5.9: Descomposición tipo tablero de ajedrez y correspondencia de las fronteras fantasma.

En algunas fronteras de los subdominios las condiciones de frontera serán iguales a las del problema real. La partición produce condiciones de frontera artificiales, conocidas como condiciones de frontera fantasma (*ghost*). Por ejemplo, en la figura 5.9(a), las condiciones de frontera reales de los subdominios se definen de forma similar que en el problema global (en los círculos vacios). Por otro lado, las condiciones de frontera fantasma, se deben definir en las interfases de los subdominios (en los cuadrados llenos).

Estas condiciones de frontera se obtienen de los subdominios vecinos y son valores del cálculo de la solución en un instante anterior al actual. La correspondencia de las fronteras fantasma con los puntos de cálculo de subdominios vecinos se muestra en la figura 5.9(b). En este esquema, se debe verificar que las condiciones de frontera fantasma sean las adecuadas para cada subproblema, es decir, que se cumplan las leyes de balance impuestas en la deducción de las ecuaciones que gobiernan el fenómeno.

La partición del dominio es producida por la clase Domain que determina el número de bloques que se tendrán en cada dirección de los ejes coordenados. La topología de la red es importante dado que dos subdominios vecinos no deben ser asignados a procesadores alejados entre sí dentro de la red física que conecta a los procesadores o computadoras donde se ejecute el programa. En la biblioteca MPI existe una manera de reordenar automáticamente los procesos, de tal manera que se aproveche la topología física de la red [68, 24]. En este trabajo los subdominios se distribuyen en un arreglo cartesiano de tal manera que cada subdominio conoce sus coordenadas dentro de dicho arreglo (tres coordenadas (I, J, K), cuando el problema es tridimensional). Lo anterior permite a cada subdominio de antemano conocer los subdominios vecinos con los que se debe comunicar, véase figura 5.10. Debido a este hecho, es posible utilizar comunicaciones persistentes en donde se calcula, desde un principio, los subdominios vecinos y la información que deben intercambiar. Después sólo es necesaro invocar las funciones de comunicación (envío y/o recepción). El beneficio de lo anterior es evitar la inicialización de *sockets* de comunicación cada vez que se necesite un intercambio de información. En un problema dependiente del tiempo, este proceso se puede realizar varias veces en cada paso de tiempo, por lo tanto la reducción del tiempo de ejecución es importante y proporcional al número de iteraciones. En el apéndice A se describen las funciones de la biblioteca MPI utilizadas en este trabajo.

5.5. Discusión

El problema de la crisis del software puede ser resuelto en parte, siguiendo un proceso de desarrollo de software bien especificado y que además sea iterativo e incremental. En este trabajo se propone un modelo de desarrollo de software basado en el proceso unificado. Este modelo, además de considerar los modelos matemático y discreto como actividades del proceso, se agregan la actividad de generalización y de optimización, dado que ambas son importantes en el desarrollo de software científico. También, se consideran iteraciones cortas, del estilo de la programación extrema, de tal manera que en cada iteración sólo se resuelvan problemas simples, agregando clases y/o módulos con bajo acoplamiento y alta cohesión.

La arquitectura del sistema esta basada en el proceso general de una simulación numérica y de ahí se obtienen tres componentes principales. Estas componentes tienen un bajo acoplamiento entre ellas y las clases contenidas en éstas tienen alta cohesión. Por ejemplo, la clase EnergyEquation<>, implementa la ecuación de energía y sólo nos preocupa definir los parámetros adecuados de acuerdo a la descripción dada en el capítulo

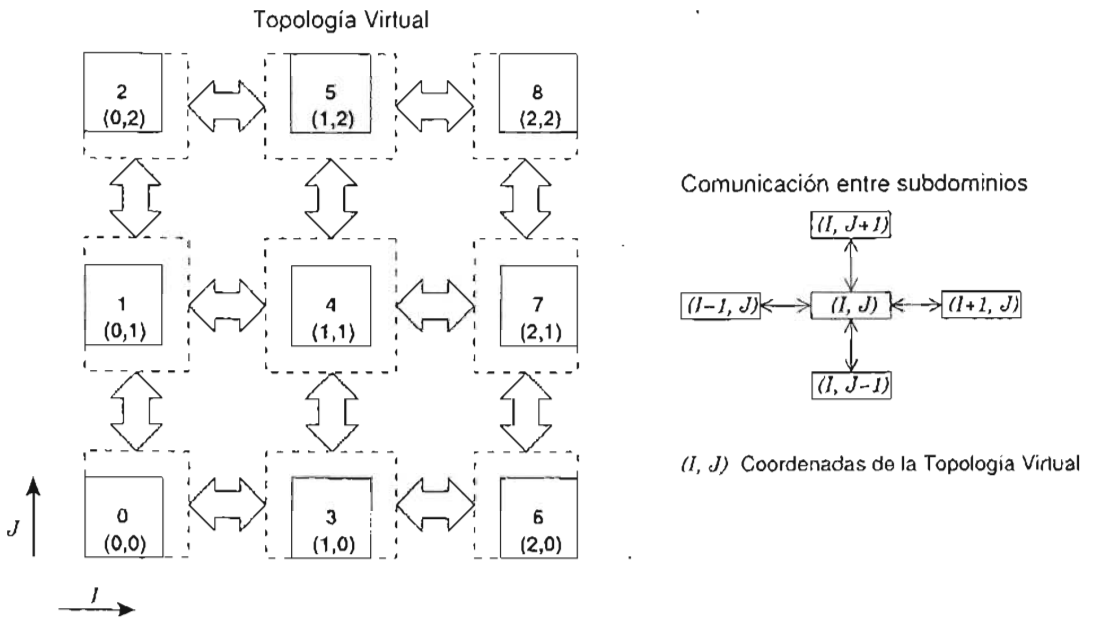


Figura 5.10: Topología Virtual (TV) en 2D generada por la partición por bloques. Se muestra la numeración de los subdominios en términos de las coordenadas de la TV.

3. Además, estas clases son polimórficas de tal manera que se pueden adaptar al esquema numérico deseado. Ejemplos de uso de ésta y otras clases se muestran en el capítulo siguiente.

El estilo de paralelización usada en este trabajo se conoce como de grano grueso y sólo es conveniente para un número reducido de subdominios. La ventaja es que, en cada subdominio se puede utilizar un algoritmo paralelo, de grano fino, para resolver el sistema lineal de ecuaciones local y de esta forma reducir aún más el tiempo de cálculo. La combinación de paralelismo de grano grueso con paralelismo de grano fino se conoce como multiparalelismo o paralelismo de varios niveles [53]. Aunque esta estratégia no se empleó en este trabajo es posible adicionarla fácilmente. Otro punto importante es que, mediante la estrategia de paralelización que hemos seguido aquí, es posible en un futuro hacer uso de la tecnología de GRID [22].

Capítulo 6

Ejemplos y resultados numéricos

En este capítulo se explica como usar y combinar cada una de las clases y funciones construidas en este trabajo para resolver diferentes problemas. Se presentan primero algunos resultados preliminares con el objetivo de explicar en detalle la forma de uso de las clases. Después se realizan algunas comparaciones del problema de convección natural en régimen laminar con *benchmarks* conocidos y se analiza un problema interesante de flujo de mezclado. Posteriormente, se muestran resultados de convección natural en régimen turbulento en tres dimensiones. Finalmente, se hace un estudio de los beneficios del paralelismo en este tipo de problemas.

En apéndice A se describen las clases construidas en este trabajo y algunos detalles de la implementación.

6.1. Resultados preliminares

El conjunto de clases desarrollado en este trabajo puede usarse fácilmente para resolver diferentes tipos de problemas numéricos. En esta primera versión sólo se consideraron problemas descritos en dominios rectangulares con mallas cartesianas. La ventaja de nuestro desarrollo es que para resolver la mayoría de los problemas que presentamos se relizan los mismos pasos:

1. Incluir los encabezados de las clases que se deseen usar.
2. Declarar e inicializar las variables del problema.
3. Declarar el sistema lineal para almacenar los coeficientes de la discretización. Este sistema puede ser compartido cuando haya más de una ecuación por resolver.
4. Declarar y definir la malla del dominio.
5. Declarar y definir las variables dependiente (campos escalares y vectoriales sobre la malla) necesarias y las condiciones iniciales.
6. Declarar y definir las ecuaciones que se deben resolver.

7. Resolver la ecuación: calcular los coeficientes del sistema lineal y resolverlo.

En los ejemplos que siguen se muestra como se realiza cada uno de los pasos.

6.1.1. Difusión en una y dos dimensiones

El primer problema que presentamos es muy simple y tiene como objetivo describir el uso de las clases y módulos desarrollados en este trabajo. Posteriormente, se hará referencia a esta descripción cuando se revisen problemas más complejos. En este caso se considera el problema de difusión de calor en una dimensión, el cual se describe en la figura 6.1. En este caso la ecuación a resolver se escribe como:

$$\frac{\partial T}{\partial t} = \nabla^2 T. \tag{6.1}$$

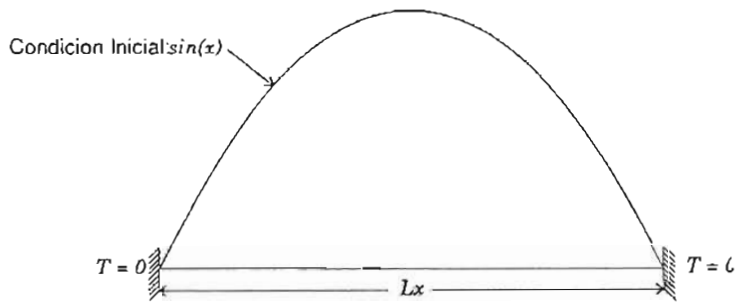


Figura 6.1: Condiciones iniciales y de frontera para el problema de difusión en 1D.

El siguiente código resuelve la ecuación (6.1), en una dimensión y con las condiciones iniciales y de frontera mostradas en la figura 6.1:

```
1  #include "Utils/Common.h"
2  #include "Equations/EnergyEquation.h"
3  #include "Meshes/StructuredMesh.h"
4  #include "NumSchemes/Diffusion.h"
5  #include "Solver/Solver.h"
6
7  int main()
8  {
9      double longitud, dt, dx, tolerancia, error, Tleft, Tright;
10     int num_nodos, num_vols, iteracion, max_iter;
11 // ... inicializacion de variables ...
12
13     DiagonalMatrix<Tri>       A(num_vols);              // Matriz A
14     ScalarField1D             b(num_vols);              // Vector b
15     StructuredMesh<double, 1> mesh(longitud, num_nodos); // Malla
16     ScalarField1D             T(mesh.getExtentVolumes());// Campo escalar T
```

```
17
18      dx = mesh.getDelta(0);
19      firstIndex index;                       //
20      T = sin(PI * index * dx / longitud);    // Condiciones
21      T(T.lbound(firstDim)) = Tleft;          // iniciales
22      T(T.ubound(firstDim)) = Tright;         //
23
24      EnergyEquation<double, 1, Diffusion<double, 1> > energia(T);
25      energia.setLinearSystem(A, b);
26      energia.setGamma(1.0);
27      energia.setDeltas(mesh.getDeltas());
28      energia.setDeltaTime(dt);
29      energia.setDirichlet(LEFT_WALL, Tleft);
30      energia.setDirichlet(RIGHT_WALL, Tright);
31
32      while (error > tolerancia && ++iteracion < max_iter) {
33          energia.calcCoefficients();
34          Solver::TDMA(energia);
35          error = energia.calcError();
36          energia.update();
37          Output::printToFile_GP(T, iteracion, "temp.", dx);
38      }
39      return 0;
40  }
```

El código de arriba se explica a continuación:

Líneas 1–5 : En estas líneas se incluyen los archivos que contienen la definición de las clases que se necesitan para resolver el problema. En este caso se incluyen los encabezados: `Common.h`, contiene definiciones de clases comunes y hace una liga con la biblioteca Blitz++ para el manejo de arreglos multidimensionales; `EnergyEquation.h`, implementa la clase polimórfica para la ecuación de energía; `StructuredMesh.h`, implementación de la clase para mallas uniformes estructuradas; `Diffusion.h`, implementación del esquema numérico que es usado como un adaptador para la ecuación de energía; `Solver.h`, implementación de los algoritmos de solución del sistema lineal. Cada uno de estos encabezados se encuentran en módulos diferentes, los cuales representan las componentes principales del sistema: el modelo matemático, las técnicas de discretización y la solución de los sistemas lineales, como se describe en la figura 5.5.

Línea 13–14 : Declaración del sistema lineal. Se declara primero un objeto de la clase `DiagonalMatrix`. En este caso, el objeto `A` es una matriz tridiagonal (nótese el parámetro `Tri` en la declaración) que contiene el sistema lineal de ecuaciones. La matriz es de tamaño `num_vols`× `num_vols`. La declaración para problemas en dos y tres dimensiones se hace de manera similar usando como parámetro

Penta o Hepta respectivamente. Después, en la línea 14 se declara un arreglo unidimensional que contendrá el lado derecho del sistema lineal.

Línea 15 : Declaración del objeto `mesh` que representa la malla del problema. En la declaración se usan los parámetros `double` y `1` que determinan la precisión y la dimensión del problema respectivamente. El objeto `mesh` toma como argumentos para su creación la longitud y el número de puntos en cada dirección.

Línea 16 : Declaración del objeto `T` que es un arreglo en donde se almacenará la solución del problema. El tamaño del arreglo se determina a partir de los datos de la malla, que se obtienen apartir del objeto `mesh`.

Líneas 18–22: Definición de las condiciones iniciales. Se utilizan las bondades de la biblioteca Blitz++, para inicializar arreglos usando operadores sobrecargados. En este caso el rendimiento no se ve afectado dado que Blitz++ utiliza *expression templates* [83], para eliminar fuentes de bajo rendimiento.

Línea 24: Se define el objeto `energia` que representa la ecuación 6.1. Se utilizan los parámetros: `double` para la precisión, `1` para la dimensión del problema y `Diffusion<double, 1>` para el esquema numérico. Este último parámetro determina la forma en que se calcularán los coeficientes de la ecuación discreta. Esta es una de las características más importantes de nuestro sistema: para cambiar de esquema numérico sólo se cambia este parámetro, pues la clase `EnergyEquation<>` es una clase polimórfica. En este caso el adaptador es `Diffusion<double, 1>`. En el apéndice A se presentan y explican todos los esquemas que se han implementado y se describe como agregar otros de forma simple. El objeto `energia` recibe como parámetro de creación el objeto `T` que es el campo donde se almacenará la solución.

Líneas 25–30 : En estas líneas se define el sistema lineal (el objeto `energia` hará referencia a los objetos `A` y `b` para almacenar el sistema lineal, de tal manera que no se aloja más memoria); el valor de Γ de la ecuación 6.1, que en este caso vale 1; el tamaño de la malla y el paso en el tiempo. Las condiciones de frontera se especifican en las líneas 29 y 30, que en este caso son dos condiciones tipo Dirichlet en los extremos. Cuando sea necesario resolver más de una ecuación, estás puede compartir el sistema lineal (`A` y `b`) para ahorrar espacio en memoria.

Líneas 32–38: Se inicia un ciclo que finaliza hasta que el error sea menor que una tolerancia especificada o se haya alcanzado el numéro máximo de iteraciones. Dentro de este ciclo primero se calculan los coeficientes, en donde se utiliza el esquema definido en la línea 24. Luego se resuelve el sistema usando el algoritmo `TDMA` que está contenido en el módulo `Solver`, finalmente se calcula el error y se actualiza la solución. El objeto `energia` tiene una copia del campo `T` que contiene los datos más recientes de la solución. Por otro lado, `T` contiene la solución en el paso anterior. Por esta razón, es necesario hacer una actualización a través del objeto

energia. La línea 37 indica que se almacene la solución en un archivo y se utiliza la función `printToFile_GP()` del módulo `Output` que recibe el campo a almacenar, el número de iteración, la cadena base para el nombre del archivo y el tamaño de la malla.

Lo que es notable del código anterior es que, siempre se seguirá el mismo formato para resolver otro tipo de problemas. Los cambios serán mínimos y básicamente serán en la dimensión del problema, el número de ecuaciones a resolver y/o el esquema numérico a utilizar.

Para este ejemplo, se utilizó una malla unidimensional de 50 puntos y un paso en el tiempo de 10^{-2}. El error permitido fue de 10^{-5}, el cual se obtiene después de 94 pasos temporales. El error RMS de la solución se muestra en la figura 6.2

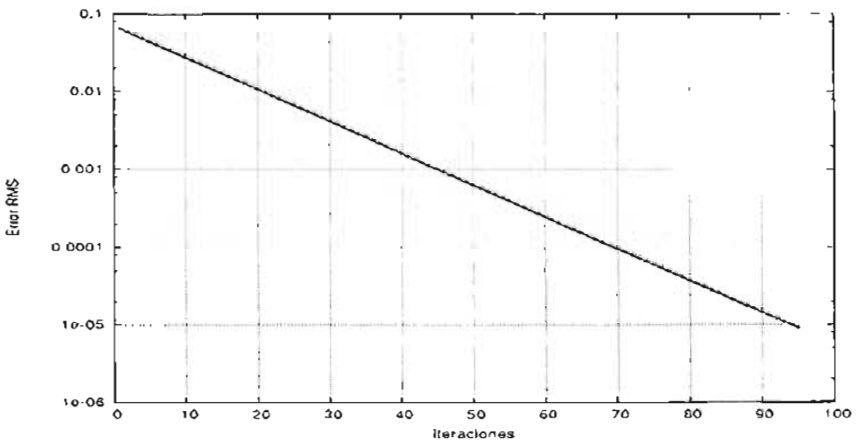


Figura 6.2: Error RMS del problema de difusión en 1D.

Para demostrar la versatilidad de nuestro sistema, en el siguiente problema se resuelve la ecuación de Laplace en dos dimensiones. El dominio de estudio es un cuadrado unitario, con las siguientes condiciones de frontera

$$T(0,y) = T(1,y) = T(x,0) = 0 \quad y \quad T(x,1) = 10\sin(\pi x)$$

En este caso se conoce la solución exacta que se escribe como sigue:

$$Ta(x,y) = 10\sin(\pi x)\frac{\exp^{\pi y} - \exp^{-\pi y}}{\exp^{\pi} - \exp^{-\pi}} \tag{6.2}$$

En este ejercicio se utiliza un código similar al anterior, con los siguientes cambios:

- En dos dimensiones el sistema lineal de ecuaciones es pentadiagonal, además se deben definir la longitud y el número de nodos en las direcciones x y y, y se deben utilizar campos escalares bidimensionales:

```
DiagonalMatrix<Penta>      A(num_vols_x, num_vols_y);
ScalarField2D              b(num_vols_x, num_vols_y);
StructuredMesh<double, 2> malla(long_x, num_nodos_x, long_y, num_nodos_y);
ScalarField2D              T(malla.getExtentVolumes());
```

En este caso el tamaño de la matriz es de $(\texttt{num\_vols\_x} \times \texttt{num\_vols\_y})^2$

- La definición de la condición inicial es muy clara dado que se está usando sobrecarga de operadores:

```
Range I(T.lbound(firstDim)+1,T.ubound(firstDim)-1);
int ej = T.ubound(firstDim);
T(I,ej) = 10 * sin (PI * dx * I);
```

- La ecuación y el esquema numérico se parametrizan con dimensión 2:

```
EnergyEquation<double, 2, Diffusion<double, 2> > energia(T);
```

- Las condiciones de frontera se definen en las cuatro paredes del cuadrado unitario:

```
energia.setDirichlet(TOP_WALL);
energia.setDirichlet(LEFT_WALL, 0.0);
energia.setDirichlet(RIGHT_WALL, 0.0);
energia.setDirichlet(BOTTOM_WALL, 0.0);
```

En la primera de estas condiciones no se utiliza ningún valor de frontera, por lo que la función `setDirichlet()` utiliza los valores de la condición inicial para definir la condición de frontera en la pared superior.

- La ecuación se resuelve usando el algoritmo descrito en la sección 4.5.2, que es una extensión del TDMA a dos dimensiones:

```
Solver::solveByLines(energia, tolerancia, tdma_iter);
```

La función `solveByLines()` recibe como argumento la ecuación a resolver, la tolerancia y el número máximo de iteraciones.

La solución numérica, así como el error con respecto a la solución exacta y el residuo, se muestran en la figura 6.3.

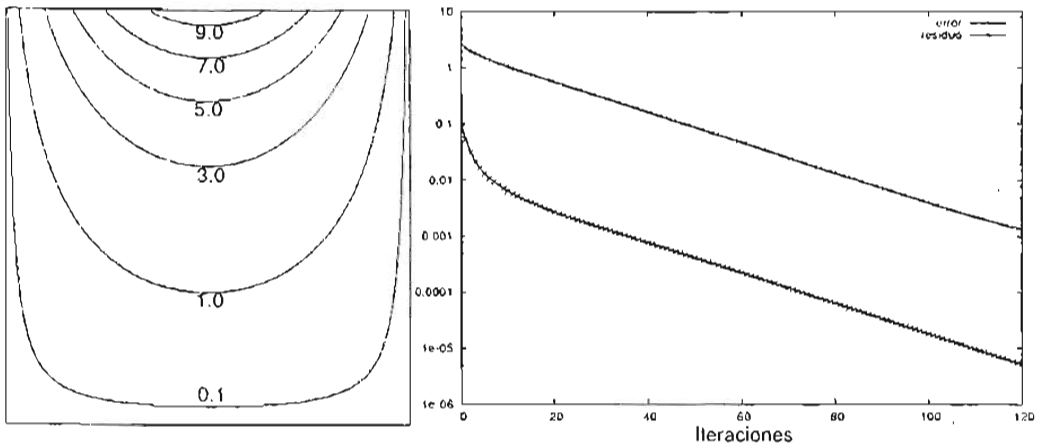


Figura 6.3: (a) Solución de la ecuación de Laplace en dos dimensiones para una malla de 64^2. (b) Se muestra el error $= \sqrt{\frac{\sum \|Ta-T\|}{N}}$, donde N = número de puntos y el residuo. El error porcentual máximo después de 120 iteraciones es de 0.03 %.

6.1.2. Convección forzada en 2D y 3D

En los siguiente ejemplos se resuelve la siguiente ecuación:

$$\frac{\partial T}{\partial t} + \vec{U}\nabla T = \nabla^2 T. \tag{6.3}$$

en donde la velocidad $\vec{U} = (u, v, w)$ es conocida y tiene la forma:

$$\vec{U} = (-A\cos(\pi y)\sin(\pi x), A\sin(\pi y)\cos(\pi x), 0) \tag{6.4}$$

que representa una celda convectiva que gira en el sentido horario y además cumple con la ecuación de continuidad. Las condiciones de frontera son las que se muestran en la figura 6.4.

Para resolver este problema necesitamos considerar lo siguiente:

- La ecuación (6.3) contiene un término convectivo, el cual debe tratarse con alguno de los esquemas numéricos descritos en la sección 4.3. Las clases donde se implementan estos esquemas son adaptadores de las ecuaciones discretas y para usarlos se debe incluir alguno de los siguientes encabezados:

  - `#include "NumSchemes/QuickE.h"`, equema QUICK.
  - `#include "NumSchemes/UpwindE.h"`, esquema Upwind.
  - `#include "NumSchemes/CentralDiffE.h"`, esquema CDS.

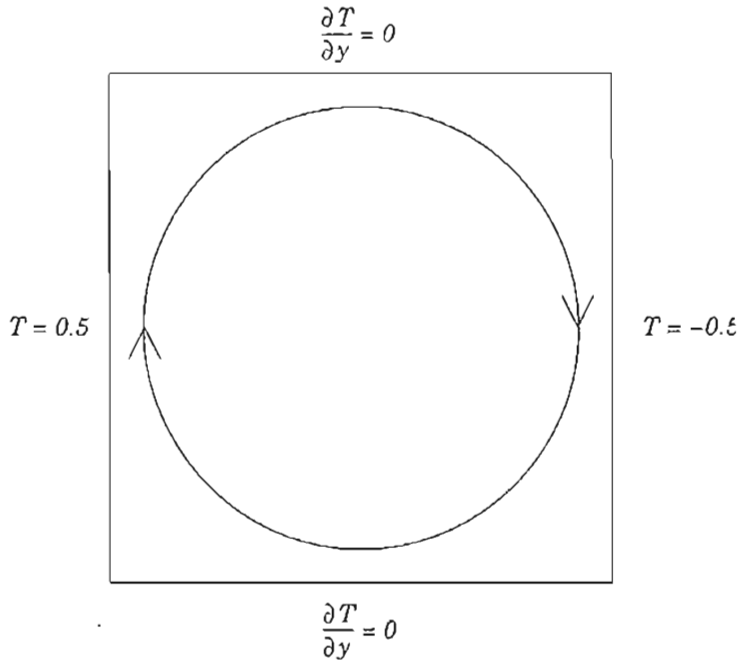


Figura 6.4: Dominio de estudio y condiciones de frontera para el problema de convección forzada en 2D.

- Las declaraciones de la matriz del sistema, la malla y los campos escalares se hacen como sigue:

```
DiagonalMatrix<Penta>      A(num_vols_x, num_vols_y);
ScalarField2D              b(num_vols_x, num_vols_y);
StructuredMesh<double, 2> malla(longitud_x, num_nodos_x,
                                longitud_y, num_nodos_y);

ScalarField2D T(malla.getExtentVolumes());     // Temperatura
ScalarField2D u(malla.getExtentVolumes());     // u-velocidad
ScalarField2D v(malla.getExtentVolumes());     // v-velocidad
ScalarField2D us(num_nodos_x, num_vols_y); // u-velocidad (staggered)
ScalarField2D vs(num_vols_x, num_nodos_y); // v-velocidad (staggered)
```

  donde los campos escalares us y vs se utilizan para almacenar las velocidades en mallas desplazadas. Nótese que estos campos escalares se definen de forma diferente a los casos anteriores: en la dirección del desplazamiento las variables están en las caras, que corresponden a los nodos, mientras que la otra dirección las variables están al centro de los volúmenes.

- Para definir las condiciones iniciales hacemos uso de las ventajas de la biblioteca Blitz++, y las calculamos de la siguiente forma:

```
const int bi = T.lbound(firstDim) + 1, ei = T.ubound(firstDim) - 1,
          bj = T.lbound(secondDim) + 1, ej = T.ubound(secondDim) - 1;
Range I(bi,ei), J(bj,ej), all = Range::all();
double dx = malla.getDelta(0);
double dy = malla.getDelta(1);

T(bi-1, all) = left_wall;
T(ei-1, all) = right_wall;
u(I,J) = -A * cos (PI * dy * J) * sin (PI * dx * I);
v(I,J) =  A * sin (PI * dy * J) * cos (PI * dx * I);
```

donde se utilizan rangos, I y J, para inicializar de manera simple y clara las componentes de la velocidad.

- Durante el cálculo, las velocidades estarán desplazadas, como se describe en la sección 4.4.1, por lo tanto, los valores almacenados en los campos escalares u y v se deben interpolar. Esto lo realizamos usando las funciones `staggerX()` y `staggerY()` del módulo `NumUtils`. En el código se escribe:

```
us = NumUtils::staggerX(u);
vs = NumUtils::staggerY(v);
```

- La declaración y definición de la ecuación a resolver se realiza como sigue:

```
EnergyEquation<double, 2, QuickE<double,2> > energia(T);
energia.setLinearSystem(A,b);
energia.setGamma(1.0);
energia.setDeltas(malla.getDeltas());
energia.setDeltaTime(dt);
energia.setNeumann(TOP_WALL, 0.0);
energia.setNeumann(BOTTOM_WALL, 0.0);
energia.setDirichlet(LEFT_WALL, left_wall);
energia.setDirichlet(RIGHT_WALL, right_wall);
energia.setUvelocity(us);
energia.setVvelocity(vs);
```

en donde se han definido el sisteman lineal, el coeficiente de difusión Γ, el tamaño de la malla, el paso en el tiempo, dos condiciones tipo Neumann en las paredes superior (TOP) e inferior (BOTTOM) y dos condiciones tipo Dirichlet en las paredes derecha (RIGHT) e izquierda (LEFT). Además se definen las velocidades que se usarán, y que en este caso serán las desplazadas.

- El ciclo para resolver el problema es el siguiente:

```
while ( (error > tolerancia) && (iteracion <= nmax) ) {
    energia.calcCoefficients();
    Solver::solveByLines(energia, tolerancia, max_iter);
    error = energia.calcError();
    residuo = energia.calcResidual();
    energia.update();
    //... actualizaciones, etc.
}
```

El problema en tres dimensiones se resuelve de forma idéntica, sólo es necesario hacer todas las declaraciones en 3D (sistema lineal, malla, campos escalares, acceso a elementos).

En dos dimensiones, para una malla de 24^2, se hizo una comparación entre los tres esquemas implementados. El resultado se presenta en la figura 6.5 en donde se muestran contornos de temperatura y una comparáción entre los tres esquemas en $y = 0{,}5$.

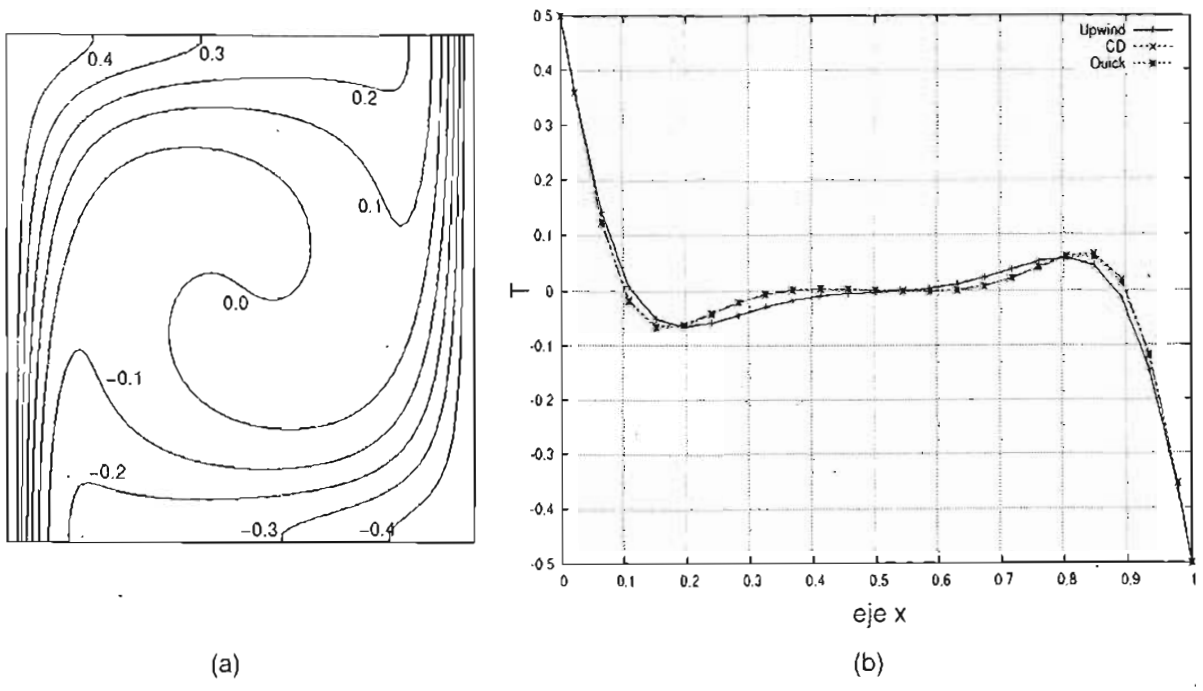


Figura 6.5: (a) Contornos de temperatura. (b) Corte en $y = 0{,}5$, se observa que para los esquemas CD y QUICK el resultado es muy similar.

6.2. Convección natural: régimen laminar

En este ejemplo se muestra la solución de un problema de convección natural en un prisma cúbico. Las paredes verticales se mantienen a una diferencia de temperatura constante, mientras que las paredes horizontales son adiabáticas. Las ecuaciones que se deben resolver son (3.12), (3.13) y (3.14). En este caso se utiliza el algoritmo SIMPLEC para resolver el problema no lineal y desacoplar la presión y velocidad de las ecuaciones de Navier-Stokes. En el código que sigue, se muestran las partes más importantes para resolver este problema en tres dimensiones (en 2D el código es similar) :

```
1   #include "Equations/EnergyEquation.h"
2   #include "Equations/XmomentumEquation.h"
3   #include "Equations/YmomentumEquation.h"
4   #include "Equations/ZmomentumEquation.h"
5   #include "Equations/PressureEquation.h"
6   #include "NumSchemes/CentralDiffE.h"
7   #include "NumSchemes/CentralDiffX.h"
8   #include "NumSchemes/CentralDiffY.h"
9   #include "NumSchemes/CentralDiffZ.h"
10  //...
11  template<class T_energy, class T_xmom, class T_ymom, class T_zmom, class T_press>
12  double SIMPLEC(T_energy &, T_xmom &, T_ymom &, T_zmom &, T_press &, int, double);
13  //...
14  int main()
15  {
16  //...
17
18      DiagonalMatrix<Hepta> A(num_vols_x, num_vols_y, num_vols_z);
19      ScalarField3D         b(num_vols_x, num_vols_y, num_vols_z);
20      StructuredMesh<double, 3> malla(longitud_x, num_nodos_x,
21                                      longitud_y, num_nodos_y,
22                                      longitud_z, num_nodos_z);
23      ScalarField3D T(malla.getExtentVolumes());                  // Temperature
24      ScalarField3D p(malla.getExtentVolumes());                  // pressure
25      ScalarField3D pp(malla.getExtentVolumes());                 // pressure corr.
26      ScalarField3D u(malla.getExtentVolumes());                  // u-velocity
27      ScalarField3D v(malla.getExtentVolumes());                  // v-velocity
28      ScalarField3D w(malla.getExtentVolumes());                  // w-velocity
29      ScalarField3D us(num_nodos_x, num_vols_y , num_vols_z);     // u staggered
30      ScalarField3D vs(num_vols_x , num_nodos_y, num_vols_z);     // v staggered
31      ScalarField3D ws(num_vols_x , num_vols_y , num_nodos_z);    // w staggered
32
33      Range all = Range::all();
34      T(T.lbound(firstDim), all, all) = left_wall;                // Left
35      T(T.ubound(firstDim), all, all) = right_wall;               // Rigth
36
```

```
37      EnergyEquation<double, 3, CentralDiffE<double,3> > energia(T);
38      energia.setLinearSystem(A,b);
39      energia.setGamma(1.0);
40      energia.setDeltas(malla.getDeltas());
41      energia.setDeltaTime(dt);
42      energia.setNeumann(TOP_WALL, 0.0);
43      energia.setNeumann(BOTTOM_WALL, 0.0);
44      energia.setDirichlet(LEFT_WALL, left_wall);
45      energia.setDirichlet(RIGHT_WALL, right_wall);
46      energia.setNeumann(FRONT_WALL, 0.0);
47      energia.setNeumann(BACK_WALL, 0.0);
48      energia.setUvelocity(us);
49      energia.setVvelocity(vs);
50      energia.setWvelocity(ws);
51
52      XmomentumEquation<double, 3, CentralDiffX<double, 3> > momento_x(us);
53      momento_x.setLinearSystem(A,b);
54      momento_x.setGamma(Prandtl);
55      momento_x.setDeltas(malla.getDeltas());
56      momento_x.setDeltaTime(dt);
57      momento_x.setUnderRelaxation(alpha_u);
58      momento_x.setDirichlet(LEFT_WALL, 0);
59      momento_x.setDirichlet(RIGHT_WALL, 0);
60      momento_x.setDirichlet(TOP_WALL, 0);
61      momento_x.setDirichlet(BOTTOM_WALL, 0);
62      momento_x.setDirichlet(FRONT_WALL, 0);
63      momento_x.setDirichlet(BACK_WALL, 0);
64      momento_x.setVvelocity(vs);
65      momento_x.setWvelocity(ws);
66      momento_x.setPressure(p);
67
68      YmomentumEquation<double, 3, CentralDiffY<double, 3> > momento_y(vs);
69      momento_y.setLinearSystem(A,b);
70      momento_y.setGamma(Prandtl);
71      momento_y.setDeltas(malla.getDeltas());
72      momento_y.setDeltaTime(dt);
73      momento_y.setUnderRelaxation(alpha_v);
74      momento_y.setDirichlet(LEFT_WALL, 0);
75      momento_y.setDirichlet(RIGHT_WALL, 0);
76      momento_y.setDirichlet(TOP_WALL, 0);
77      momento_y.setDirichlet(BOTTOM_WALL, 0);
78      momento_y.setDirichlet(FRONT_WALL, 0);
79      momento_y.setDirichlet(BACK_WALL, 0);
80      momento_y.setUvelocity(us);
81      momento_y.setWvelocity(ws);
82      momento_y.setPressure(p);
```

```
83      momento_y.setTemperature(T);
84      momento_y.setRayleigh(Rayleigh);
85
86      ZmomentumEquation<double, 3, CentralDiffZ<double, 3> > momento_z(ws);
87      momento_z.setLinearSystem(A,b);
88      momento_z.setGamma(Prandtl);
89      momento_z.setDeltas(malla.getDeltas());
90      momento_z.setDeltaTime(dt);
91      momento_z.setUnderRelaxation(alpha_w);
92      momento_z.setDirichlet(LEFT_WALL, 0);
93      momento_z.setDirichlet(RIGHT_WALL, 0);
94      momento_z.setDirichlet(TOP_WALL, 0);
95      momento_z.setDirichlet(BOTTOM_WALL, 0);
96      momento_z.setDirichlet(FRONT_WALL, 0);
97      momento_z.setDirichlet(BACK_WALL, 0);
98      momento_z.setUvelocity(us);
99      momento_z.setVvelocity(vs);
100     momento_z.setPressure(p);
101
102     PressureEquation<double, 3 > presion(pp);
103     presion.setLinearSystem(A,b);
104     presion.setDeltas(malla.getDeltas());
105     presion.setDeltaTime(dt);
106     presion.setUnderRelaxation(alpha_p);
107     presion.applyBounds(1, num_nodos_x-1, 1, num_nodos_y-1, 1, num_nodos_z-1);
108     presion.setUvelocity(us);
109     presion.setVvelocity(vs);
110     presion.setWvelocity(ws);
111     presion.setPressure(p);
112 //...
113     for(int iteracion = 1; iteracion <= iteraciones_max; ++iteracion) {
114         sorsum = SIMPLEC(energia, momento_x, momento_y, momento_z, presion,
115                          max_iter, tolerancia);
116         if( iteracion % frecuencia == 0 ) {
117             Output::printToFile_DX(T, iteracion , "temp.");
118             Output::printToFile_DX(p, iteracion, "pres.");
119             NumUtils::interX(u, us);
120             NumUtils::interY(v, vs);
121             NumUtils::interZ(w, ws);
122             Output::printToFile_DX(u, v, w, iteracion, "velc.");
123         }
124     }
125     return 0;
126 }
```

Líneas 1–9 : Primero se incluyen los encabezados donde se implementan las ecuaciones

discretas para la energía, cantidad de movimiento en las tres direcciones y corrección a la presión. También se incluye el esquema numérico que se usará en cada ecuación. En este ejemplo se incluye el encabezado que implementa el esquema CDS.

Líneas 11–22: Se declara la función `SIMPLEC()`en donde se implementa el algoritmo descrito en la sección 4.4.3.

Líneas 18–31: Se declara el sistema lineal, la malla y los campos escalares necesarios para almacenar la solución. Nótese que todo se hace tomando en cuenta tres dimensiones.

Líneas 33–35: Se definen la condiciones iniciales para la temperatura.

Líneas 37–112: Declaración y definición de las cinco ecuaciones discretas que se resolverán. En cada una se indica el esquema numérico, las condiciones de frontera y los parámetros necesarios de acuerdo con la definición del problema. En este ejemplo se está utilizando el esquema CDS, de tal manera que el parámetro con el que se definen las ecuaciones en las líneas 37, 52, 68 y 86 es de la forma `CentralDiff?<double, 3>`, con `? = E, X, Y, Z`. Si se deseará utilizar el esquema QUICK, sólo es necesario modificar esas líneas y parametrizar las ecuaciones con los adaptadores `Quick?<double, 3>` e incluir los encabezados correspondientes en las líneas 6 – 9. Nótese que la definición de las ecuaciones no cambia mucho, aún cuando matemáticamente se escriben de manera muy distinta.

Líneas 114–125: El ciclo que se muestra entre estas líneas resuelve el problema durante un cierto número de iteraciones en el tiempo. Dentro de este ciclo se ejecuta la función `SIMPLEC()` que recibe como argumentos los objetos que representan las cinco ecuaciones discretas. Entre las líneas 118 y 123 se hace la impresión de los resultados, obsérvese que la velocidad se debe interpolar a los centros de los volúmenes antes de almacenarse.

La función `SIMPLEC()`, que se ejecuta en la línea 114 del código de arriba, se implementa como sigue:

```
1   template<class T_energy, class T_xmom, class T_ymom, class T_zmom,
2            class T_press>
3   double SIMPLEC(T_energy &energy, T_xmom &xmomentum, T_ymom &ymomentum,
4                  T_zmom &zmomentum, T_press &pressure,
5                  int max_iter, double tolerancia)
6   {
7       double sorsum = 10.0, tol = 1e-02;
8       int counter = 0;
9       ScalarField3D T = energy.getPhi();
10
```

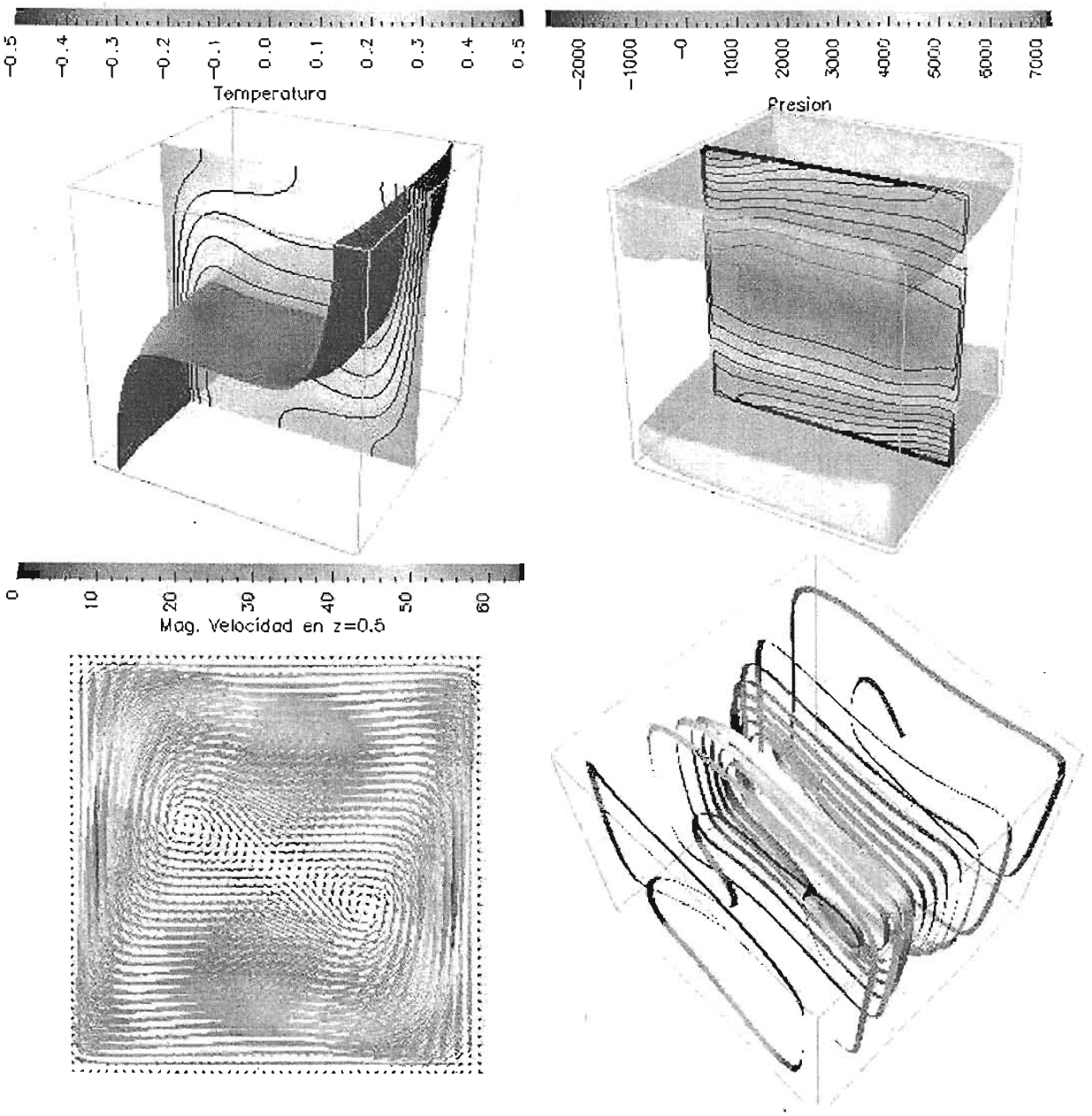


Figura 6.6: Convección natural en tres dimensiones.

6.2.1. Comparación con un *benchmark*

Para validar cuantitativamente el código se realizaron comparaciones con el *benchmark* publicado por de Vahl [14]. En este benchmark se resuelve un problema de convección natural en dos dimensiones, en donde se considera el aire como fluido de estudio

cuyo número de Prandtl es igual a 0.71. Las cantidades que se calculan y comparan son:

- U_{max} velocidad horizontal máxima en la línea $x = 0.5$.
- y coordenada vertical donde ocurre U_{max}.
- V_{max} velocidad vertical máxima en la línea $y = 0.5$.
- x coordenada horizontal donde ocurre V_{max}.
- Nu Número de Nusselt en la pared vertical $x = 0$.

El problema se resuelve para diferentes números de Rayleigh: 10^3, 10^4, 10^5 y 10^6 en una malla de 81^2. Los resultados numéricos obtenidos usando el esquema QUICK se tabulan en la tabla 6.2.1.

	Ra = 10^3		Ra = 10^4		Ra = 10^5		Ra = 10^6	
	dVD[14]	BibClas	dVD[14]	BibClas	dVD[14]	BibClas	dVD[14]	BibClas
U_{max}	3.649	3.649	16.178	16.184	34.722	34.824	64.630	65.451
y	0.813	0.814	0.823	0.827	0.855	0.864	0.850	0.852
V_{max}	3.697	3.698	19.617	19.633	68.590	68.684	219.360	218.098
x	0.178	0.179	0.119	0.117	0.066	0.068	0.0379	0.0375
Nu	1.117	1.118	2.238	2.248	4.509	4.546	8.817	8.970

Cuadro 6.1: Comparación con el benchmark publicado por de Vahl Davis [14]. Los resultados numéricos fueron obtenidos usando el esquema QUICK, en una malla de 81×81.

Los resultados de la tabla 6.2.1 se comparan muy bien con los resultados de de Vahl Davis [14]. La diferencia porcentual máxima de todas las cantidades calculadas es de 1.7%. Contornos de velocidad y temperatura, así como varias trayectorias de partículas, se muestran en la figura 6.7.

6.3. Flujo de mezclado

Un ejemplo interesante en donde se puede demostrar la versatilidad de nuestro sistema es el siguiente. En este ejercicio se presenta una técnica de mezclado mediante convección natural dependiente del tiempo. Este ejemplo es importante debido a que tiene aplicaciones en la industria farmacéutica, cristalografía, manufactura, entre otras.

Considérese un cuadrado en dos dimensiones como se muestra en la figura 6.8. La mitad izquierda de la pared superior es enfriada de manera cíclica, mientras que la mitad derecha de la pared inferior es calentada de manera similar. Las dos paredes verticales son adiabáticas. El valor de la velocidad es igual a cero en todas las paredes. En forma analítica las condiciones de frontera se escriben como sigue:

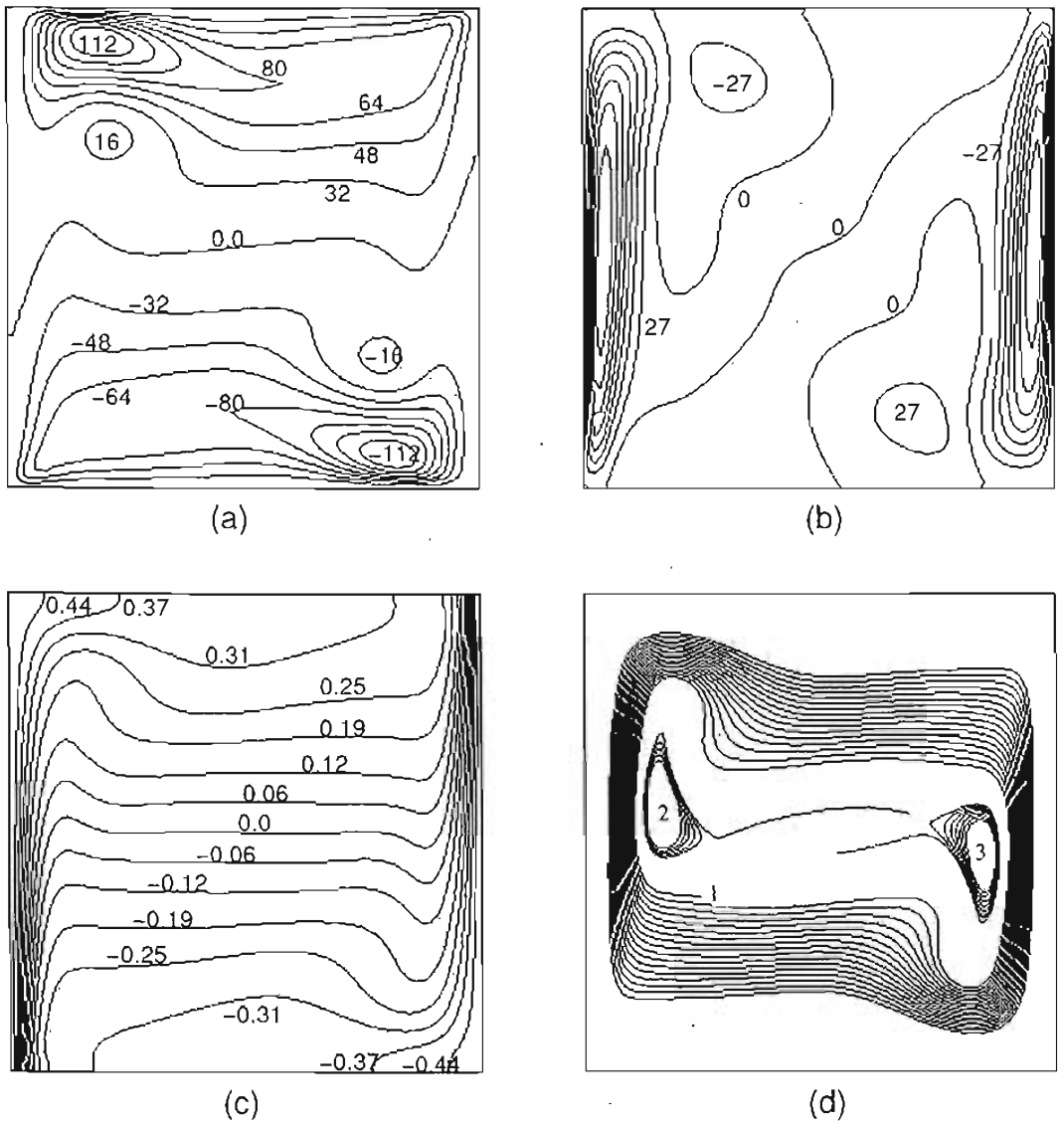


Figura 6.7: Resultado para $Ra = 10^6$ con el esquema QUICK en una malla de 81^2. (a) Componente u de la velocidad, (b) componente v de la velocidad, (c) temperatura y (d) Trayectoria de partículas: 1=(0.247,0.370), 2=(0.197,0.556), 3=(0.827,0.444).

$\mathbf{u} = 0$	y	$\partial T/\partial x = 0$	para	x =-0.5,0.5	-0.5 $\leq y \leq$ 0.5,
$\mathbf{u} = 0$	y	$T = 0$	para	-0.5 $\leq x \leq$ 0	y =-0.5,
$\mathbf{u} = 0$	y	$T = f_1(t)$	para	0 $\leq x \leq$ 0.5	y =-0.5,
$\mathbf{u} = 0$	y	$T = f_2(t)$	para	-0.5 $\leq x \leq$ 0	y =0.5,
$\mathbf{u} = 0$	y	$T = 0$	para	0 $\leq x \leq$ 0.5	y =0.5 .

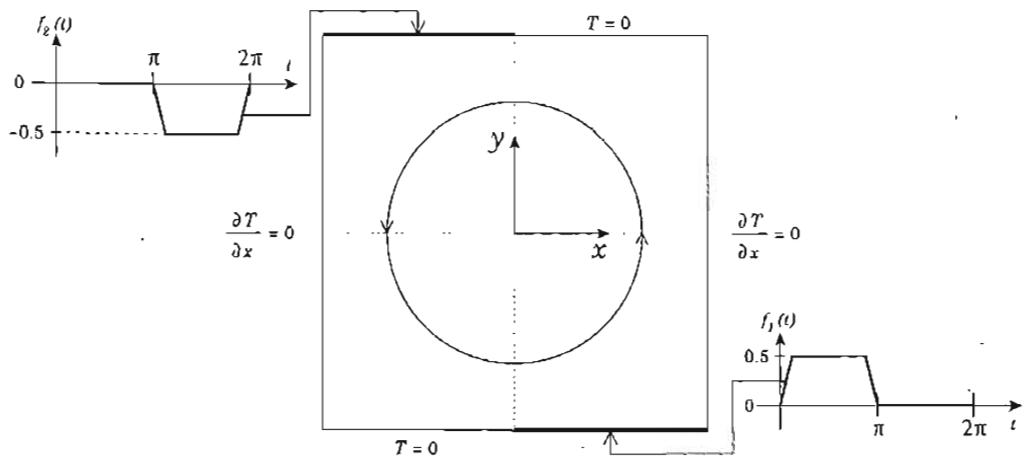


Figura 6.8: Geometría y condiciones de frontera térmicas para este ejemplo.

El primer ciclo de las funciones periódicas f_1 y f_2 está definido por las siguientes expresiones:

$$f_1(t) = \begin{cases} 0{,}5\sin^2(4t) & \text{para} \quad 0 \leq t < \pi/8 \\ 1 & \text{para} \quad \pi/8 \leq t < 7\pi/8 \\ 0{,}5\sin^2(4t - 3\pi) & \text{para} \quad 7\pi/8 \leq t < \pi \\ 0 & \text{para} \quad \pi \leq t < 2\pi \end{cases} \tag{6.5}$$

y

$$f_2(t) = \begin{cases} 0 & \text{para} \quad 0 \leq t < \pi \\ -0{,}5\sin^2(4t - 3\pi) & \text{para} \quad \pi \leq t < 9\pi/8 \\ 1 & \text{para} \quad 9\pi/8 \leq t < 15\pi/8 \\ -0{,}5\sin^2(4t - 6\pi) & \text{para} \quad 15\pi/8 \leq t < 2\pi \end{cases} \tag{6.6}$$

Nótese que la fase relativa del cambio de temperatura de las paredes es π.

Para resolver este ejemplo, se utiliza un código similar al mostrado en la sección 6.2 (adaptado para dos dimensiones). El único cambio es en el ciclo de las líneas 113–124:

```
for(iteracion = 1; iteracion <= iteraciones_max; ++iteracion) {
    boundaryConditions(T, iteracion);
    energia.update_phi(T);
    sorsum = SIMPLEC(energia, momento_x, momento_y, presion,
                     max_iter, tolerancia, dx, dy);
    //...
}
```

En este último caso, solamente se agrega la función `boundaryConditions()`, la cual calcula las condiciones de frontera en las paredes superior e inferior de la cavidad mediante las funciones (6.5) y (6.6).

Las temperatura oscilatoria impuesta en las paredes superior e inferior de la cavidad ocasionan la formación de plumas térmicas ascendentes y descendentes en regiones cercanas a las paredes verticales. Estas estructuras inducen un movimiento transitorio inicial seguido por un movimiento periódico. Los resultados que se presentan a continuación se refieren al movimiento periódico, una vez que el transitorio ha desaparecido. En este ejemplo se tomó un número de Rayleigh igual a 10^5, mientras que el número de Prandtl es igual a 5.

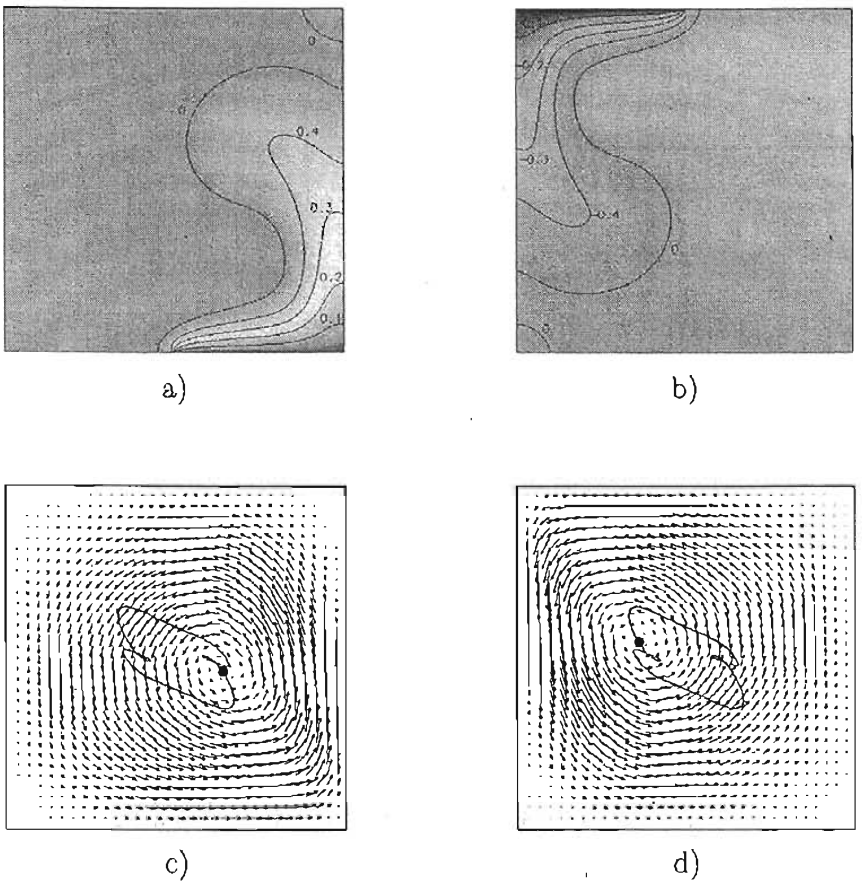
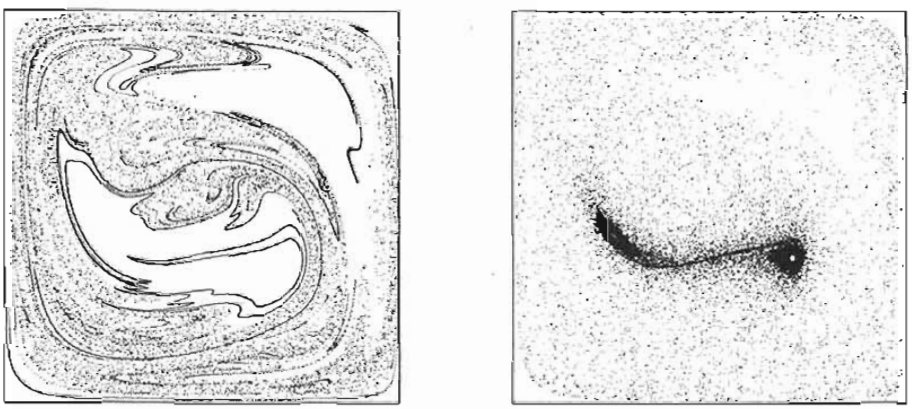


Figura 6.9: Temperature: a) $\phi = \pi/2$, b) $\phi = 3\pi/2$, Velocity vectors: c) $\phi = \pi/2$ d) $\phi = 3\pi/2$. La línea muestra la trayectoria del centro del vórtice y el punto es su posición instantánea.

Las figuras 6.9 a) y b) muestran las plumas ascendente y descendente respectivamente. En este ejemplo ϕ representa la fase dentro del ciclo de las funciones f_1 y f_2 (véase figura 6.8). La pluma ascendente y su interacción con la pared superior, generan un vórtice cuyo centro (punto con velocidad igual a cero) está desplazado hacia la derecha del centro de la cavidad, véase figura 6.9 c). En la segunda parte del ciclo, la pluma descendente genera un flujo correspondiente, véase figura 6.9 d). Debido la naturaleza periódica de la formación de las plumas, el centro del vórtice describe una

curva alrededor del centro de la cavidad. Esta trayectoria se muestra en las figuras 6.9 c) y d), y la posición instantánea del vórtice se denota con un punto.

En resumen, el efecto global del protocolo de calentamiento mostrado en la figura 6.8 es la generación de un vórtice cuyo centro rota alrededor del centro geométrico de la cavidad en una orbita cerrada. Esta es una combinación de un efecto de *blinking vortex* con un efecto de traslación-rotación. La eficiencia de mezclado se mide cualitativamente mediante el seguimiento Lagrangiano [13] de un conjunto de puntos. La figura 6.10 muestra la posición de 5×10^5 puntos, localizados originalmente en la línea y =0.5, después de 5 y 50 ciclos.



a) b)

Figura 6.10: La posición de 5×10^4 puntos originalmente localizados en la línea horizontal y =0.5 es mostrada después de a) 5 y b) 50 ciclos.

6.4. Convección natural: régimen turbulento

Para este problema se han implementado esquemas numéricos especiales y algunas funciones auxiliares para el modelo de turbulencia descrito en la sección 3.2. Estos esquemas y funciones se agrupan en el espacio de nombres LES_SSF y son clases construidas en base a la arquitectura descrita en la sección 5.1. En este caso se resolverá el problema en la geometría mostrada en la figura 6.11.

Las condiciones de frontera que se consideran para este problema son:

$$
\begin{array}{lllllll}
\mathbf{u} = 0 & \text{y} & T = 0.5 & \text{para} & x = 0 & 0 \le y \le L_y & 0 \le z \le L_z, \\
\mathbf{u} = 0 & \text{y} & T = -0.5 & \text{para} & x = L_x & 0 \le y \le L_y & 0 \le z \le L_z, \\
\mathbf{u} = 0 & \text{y} & \partial T/\partial z = 0 & \text{para} & 0 \le x \le L_x & 0 \le y \le L_y & z = 0, L_z, \\
\mathbf{u} = 0 & \text{y} & \partial T/\partial y = 0 & \text{para} & 0 \le x \le L_x & y = 0, L_y & 0 \le z \le L_z \, .
\end{array}
$$

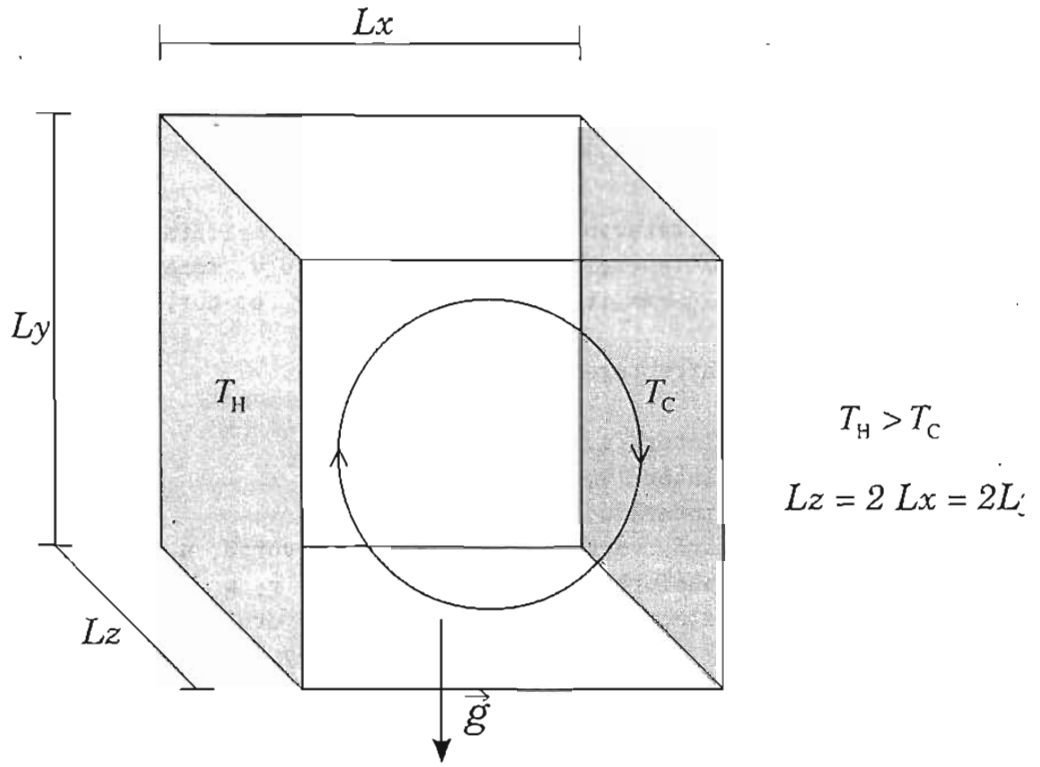


Figura 6.11: Geomtría del dominio donde se resuelve el problema de turbulencia

El código que resuelve el problema de convección turbulenta es similar al mostrado en la sección anterior, y sólamente son necesarios los siguientes cambios:

- Los encabezados que se deben agregar, suponiendo que se desea usar el esquema CDS, son los siguientes:

```
#include "LES/NumSchemes/CentralDiffE.h"
#include "LES/NumSchemes/CentralDiffX.h"
#include "LES/NumSchemes/CentralDiffY.h"
#include "LES/NumSchemes/CentralDiffZ.h"
#include "LES/les_ssf.h"
```

  Obsérvese que estos esquemas están en el directorio especial LES, en donde se implementa todo lo que tiene que ver con el modelo de turbulencia.

- Se deben declarar campos escalares para la viscosidad turbulenta y la vorticidad:

```
ScalarField3D nut(malla.getExtent());   // viscosidad turbulenta
ScalarField3D vort1(malla.getExtent()); // vorticidad en x
```

```
ScalarField3D vort2(malla.getExtent()); // vorticidad en y
ScalarField3D vort3(malla.getExtent()); // vorticidad en z
```

- El ciclo temporal se modifica de la siguiente manera:

```
 for(iteracion = 1; iteracion <= iteraciones_max; ++iteracion) {
     sorsum = SIMPLEC(energia, momento_x, momento_y, momento_z, presion,
                      max_iter, tolerancia, u, v, w, nut, dx, dy, dz);
//
// LES : Selective Structure Function
//
          NumUtils::interX(u, us);
          NumUtils::interY(v, vs);
          NumUtils::interZ(w, ws);
          DiffOperators::vorticity(vort1, vort2, vort3, u, v, w, dx, dy, dz);
          LES_SSF::calcTurbulentViscosity(nut, u, v, w, dx, dy, dz);
          LES_SSF::selectTurbulentViscosity(nut, vort1, vort2, vort3,
                                            dx, dy, dz);
     if( iteracion % frecuencia == 0 ) {
          Output::printToFile_DX(T, iteracion , "temp.");
          Output::printToFile_DX(p, iteracion, "pres.");
          Output::printToFile_DX(u, v, w, iteracion, "velc.");
          Output::printToFile_DX(vort1, vort2, vort3, iteracion, "vort.");
          Output::printToFile_DX(nut, iteracion, "nu_t.");
     }
 }
```

donde se ha agregado el cálculo de la vorticidad `vorticity()`; el cálculo de la viscosidad turbulenta `calcTurbulentViscosity()` y la selección de los lugares donde esta viscosidad tendrá efecto `selectTurbulentViscosity()`, ambas declaradas y definidas dentro del espacio de nombres LES_SSF.

- La función `SIMPLEC()` se modifica ligeramente como sigue:

```
double SIMPLEC(T_energy &energy, T_xmom &xmomentum,
               T_ymom &ymomentum, T_zmom &zmomentum,
               T_press &pressure, int max_iter, double tolerancia,
               T_field &nut, double dx, double dy, double dz )
{
//...
    while ( (sorsum > tolerance) && (counter < 20) ) {

        energy.calcCoefficients(nut);
//...
        xmomentum.calcCoefficients(nut);
```

```
//...
        ymomentum.calcCoefficients(nut);
//...
        zmomentum.calcCoefficients(nut);
//...
        pressure.calcCoefficients(xmomentum, ymomentum, zmomentum);
//...
        pressure.correctVelocity();
        pressure.correctPressure();

        sorsum = fabs( pressure.getSorsum() );

        ++counter;
    }
```

donde se observa que ahora el SIMPLEC() recibe un argumento más que es la viscosidad turbulenta y cada ecuación, al momento de calcular los coeficientes, toma como argumento esta viscosidad.

Este problema ha sido resuelto numéricamente usando diferentes modelos de turbulencia por otros autores, véase por ejemplo [31, 59]. Además, existen datos experimentales con los que se pueden comparar los resultados numéricos [75, 76]. En este caso se realizaron dos ejemplos para mallas de $64 \times 64 \times 48$ y $96 \times 96 \times 64$. Se utilizó el esquema CDS en todas las ecuaciones, pues es menos dispersivo que el Quick. La figura 6.12 muestra el número de Nusselt calculado en la pared derecha (el resultado es similar para la pared izquierda). Se observa que este número modifica su valor considerablemente de la malla gruesa :

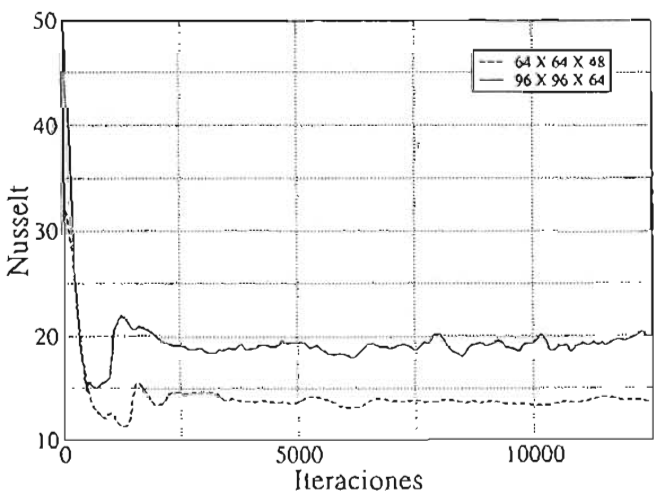


Figura 6.12: Número de nusselt para mallas de $64 \times 64 \times 48$ y $96 \times 96 \times 64$.

Para hacer una comparación con los resultados experiementales, se calculó el número de Nusselt en las cuatro paredes y se graficó el resultado en sentido horario. En la figura 6.13 los puntos representan los resultados numéricos publicados en [75] y la línea continua es el número de Nusselt que obtuvimos en una malla de $160 \times 160 \times 80$ volúmenes. Como se puede apreciar, el modelo captura la forma cualitativa de la transferencia de calor local, pero falla en la parte de los picos que se generan cerca de las esquinas. La diferencia cuantitativa que existe entre nuestros resultados y los experimentales se debe a que, estamos utilizando una malla uniforme. Para capturar y resolver la capa límite que se genera cerca de las paredes, es necesario agregar más puntos en esas zonas, sin adicionar puntos en la parte media de la cavidad. Entonces, para mejorar cuantitativamente los resultados, se debería utilizar una malla no uniforme que contenga más puntos cerca de las paredes. En este trabajo no se implementaron ese tipo de mallas, por lo que esta prueba se realizará en un trabajo futuro.

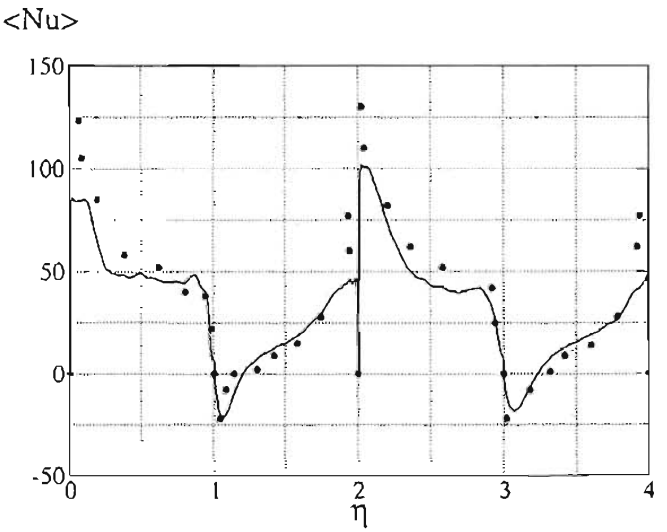


Figura 6.13: Número de nusselt promedio alrededor de las cuatro paredes para una malla de $160 \times 160 \times 80$.

Un ejemplo del flujo dentro de la cavidad se da en la figura 6.14 donde se grafica el flujo promedio y el flujo secundario en la esquina superior izquierda del plano z =0.5. La figura 6.15 muestra la distribución de la viscosidad turbulenta, véase ecuación (3.32). Las figuras 6.14 y 6.15 demuestran que la mayoría del proceso turbulento está presente en la capa límite cerca de las paredes.

6.5. Paralelismo

El sistema contiene un módulo para descomponer el dominio de estudio en varios subdominios, de tal manera que en cada una de ellos se resuelve un problema similar

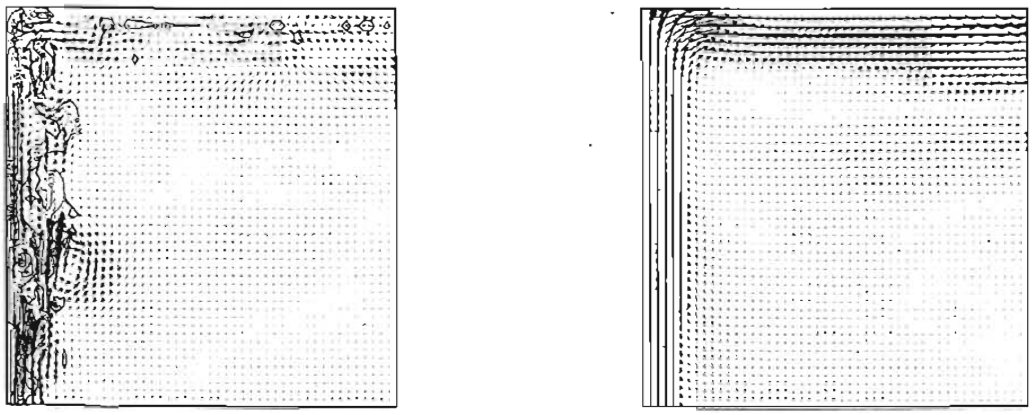


Figura 6.14: (a) Flujo secondario instantáneo e isolines de la viscosidad turbulenta ν_t. (b) flujo promedio.

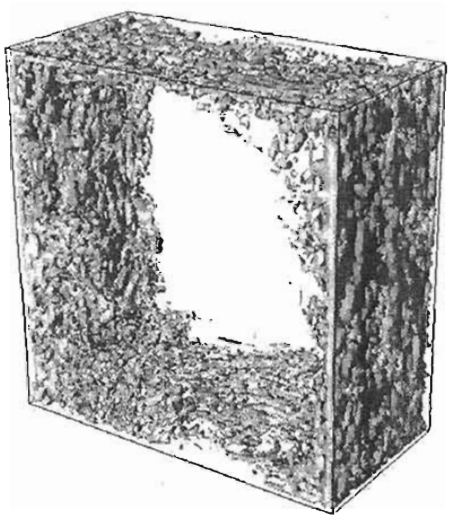


Figura 6.15: Distribución de la viscosidad turbulenta. Razón de aspecto de la cavidad 1:1:0.5, Ra = 1.59 $\times 10^9$ and Pr = 0.7.

pero agregando condiciones de frontera fantasma como se describe en la sección 5.3. En este módulo se utiliza la biblioteca de envío de mensajes MPI [24, 68] para controlar las comunicaciones entre los subdominios. Las clases de este módulo se utilizan en combinación con las clases que resuelven el problema numéricamente en forma serial. De esta manera se esta aprovechando la característica de reutilización de la POO. Para ejemplificar el uso de las clases para paralelismo, en esta sección se muestra como resolver problema de la sección 6.1.2 en paralelo. El código es similar y se muestra a

continuación:

```
1   //...
2   #include "DD/Blocks.h"
3   #include "DD/Domain.h"
4   #include "DD/SubDomain.h"
5   #include <mpi.h>
6
7   int main(int argc, char **argv)
8   {
9   //...
10      int rank, size, Irank, Jrank, Krank, bloques[2];
11      MPI::Status status;
12      MPI::Init(argc, argv);
13      rank = MPI::COMM_WORLD.Get_rank();
14      size = MPI::COMM_WORLD.Get_size();
15
16      if (rank == 0) {
17  //... Inicializacion de datos ...
18      }
19
20      MPI::COMM_WORLD.Barrier();
21      MPI::COMM_WORLD.Bcast(&longitud_x,1,MPI::DOUBLE,0);
22      MPI::COMM_WORLD.Bcast(&longitud_y,1,MPI::DOUBLE,0);
23      MPI::COMM_WORLD.Bcast(&num_nodos_x,1,MPI::INT,0);
24      MPI::COMM_WORLD.Bcast(&num_nodos_y,1,MPI::INT,0);
25  //...
26      MPI::COMM_WORLD.Barrier();
27
28      Domain<StructuredMesh<double, 2>, 2 > caja(longitud_x, num_nodos_x,
29                                                 longitud_y, num_nodos_y);
30      Blocks<2> particion(bloques[0], bloques[1]);
31      caja.decomposition(particion, rank);
32      SubDomain<StructuredMesh<double, 2>, 2>& subdominio = caja.getSubDomain();
33      Irank = subdominio.get_Irank();
34      Jrank = subdominio.get_Jrank();
35      Krank = subdominio.get_Krank();
36
37      StructuredMesh<double, 2>& submalla = subdominio.getMesh();
38      double dx = submalla.getDelta(0);
39      double dy = submalla.getDelta(1);
40
41      Range rango_x = subdominio.getLocalRanges(X);
42      Range rango_y = subdominio.getLocalRanges(Y);
43      ScalarField2D T(rango_x, rango_y);
44      ScalarField2D u(rango_x, rango_y);
```

```
45      ScalarField2D v(rango_x, rango_y);
46      ScalarField2D us(rango_x, rango_y);
47      ScalarField2D vs(rango_x, rango_y);
48
49      const int bi = T.lbound(firstDim) + 1, ei = T.ubound(firstDim) - 1,
50                bj = T.lbound(secondDim) + 1, ej = T.ubound(secondDim) - 1;
51      Range I(bi,ei), J(bj,ej), all = Range::all();
52      if (Irank == 0)            T(bi+1,all) = left_wall;
53      if (Irank == bloques[0]-1) T(ei-1,all) = right_wall;
54      u(I,J) = -A * cos (PI * dy * J) * sin (PI * dx * I);
55      v(I,J) =  A * sin (PI * dy * J) * cos (PI * dx * I);
56      NumUtils::staggerX(us, u);
57      NumUtils::staggerY(vs, v);
58
59      TinyVector<int, 2> rindex(0, 0);
60      T.reindexSelf(rindex);
61      u.reindexSelf(rindex); v.reindexSelf(rindex);
62      us.reindexSelf(rindex); vs.reindexSelf(rindex);
63
64      int svx = particion.getVolumes(X, Irank, Jrank, Krank),
65          svy = particion.getVolumes(Y, Irank, Jrank, Krank);
66      DiagonalMatrix<Penta> A(svx, svy);
67      ScalarField2D         b(svx, svy);
68
69      EnergyEquation<double, 2, QuickE<double,2> > energia(T);
70      energia.setLinearSystem(A,b);
71      energia.setGamma(1.0);
72      energia.setDeltas(submalla.getDeltas());
73      energia.setDeltaTime(dt);
74      energia.setUvelocity(us);
75      energia.setVvelocity(vs);
76      subdominio.setRealNeumann(energia, TOP_WALL, 0.0);
77      subdominio.setRealNeumann(energia, BOTTOM_WALL, 0.0);
78      subdominio.setRealDirichlet(energia, LEFT_WALL, left_wall);
79      subdominio.setRealDirichlet(energia, RIGHT_WALL, right_wall);
80      subdominio.setGhost(energia);
81      subdominio.comm_init(T, 0);
82
83      for(iteracion = 1; iteracion <= nmax; ++iteracion) {
84          energia.calcCoefficients();
85          energia.applyDirichletG2D();
86          Solver::solveByLines(energia, tolerancia, max_iter);
87          energia.update();
88
89          if (Jrank == 0)              T(all, bj-1) = T(all,bj);
90          if (Jrank == bloques[1] - 1) T(all, ej+1) = T(all,ej);
```

```
91
92          subdominio.comm_start(0);
93          energia.update_phi(T);
94      }
95      Output::printToFile_DX(T, iteracion, temp);
96
97      MPI::Finalize();
98
99      return 0;
100 }
```

Líneas 2–5 : En este caso debemos incluir los encabezados correspondientes a las clases donde está la implementación de la descomposición del problema. Además se incluye el encabezado principal de MPI.

Líneas 10–14 : Se declaran variables para obtener el rango del procesador (`rank`) y el número de procesadores activos. En la línea 12 se inicializa MPI.

Líneas 16–17 : Los datos del problema se inicializan en el procesador 0.

Líneas 20–24 : Después de que el procesador 0 ha hecho la inicialización de los datos, se envían a todos los procesadores que trabajarán en paralelo (se realiza un *broadcasting*). Este proceso de envío se encierra entre dos barreras (`MPI::COMM_WORLD.Barrier()`) para sincronizar el inicio del proceso sin pérdida ni sobreescritura de datos.

Líneas 28–35 : Posterioremente se declara un objeto de tipo `Domain`, que recibe como parámetro el tipo de malla que usará. Se declara también una partición, `Blocks`, que recibe como argumentos el número de bloques que habrá en cada eje coordenado. En la línea 31, se aplica la función `decomposition()` del objeto que representa el dominio global y recibe como argumento el tipo de partición y el rango del procesador que está realizando la partición. Todos los procesadores realizan esta acción y automátricamente se genera una topología virtual, como se describió en la sección 5.3, en donde cada uno de los subdominios conoce sus coordenadas. Estas coordenadas se obtienen en las líneas 33–35. En la línea 32 se obtiene el subdominio que le toca a cada procesador. El objeto `subdominio` se encarga de realizar diferentes inicializaciones.

Líneas 37–39 : Aquí se obtiene la malla a partir del objeto `subdominio`. Esta malla es en realidad una submalla que discretiza el subdominio en cuestión.

Líneas 41–57 : Se hace la inicialización de los campos que se usarán durante el cálculo. Obsérvese que todo se hace en términos de los subdominios.

Líneas 59–62 : Debido a que reusaremos las clases para la solución del problema serial, se hace en estas líneas una reindexación de los campos para que no haya problemas con los cálculos.

Líneas 64–81 : Se declara el sistema lineal y se define la ecuación a resolver de manera similar que en el caso serial. La diferencia es que ahora las condiciones de frontera son determinadas por cada subdominio. De esta manera, dado que cada subdominio conoce su posición, se determinan automáticamente los lugares donde se deben imponer condiciones de frontera reales y condiciones de frontera fantasma. Todo lo hace le objeto `subdominio`. La línea 81 es importante dado que inicializa los *sockets* de comunicación entre los subdominios, esto se hace una sola vez mediante operaciones persistentes de MPI para no afectar el rendimiento del código.

Líneas 83–94 : Se inicia un ciclo para resolver en el tiempo. En la línea 85 se deben aplicar las condiciones de frontera fantasma explícitamente. En las líneas 92 y 93 se inicia la comunicación entre los procesadores y se actualiza la solución en cada uno de ellos.

Como se puede observar, los cambios con respecto al código serial son mínimos. Además, las operaciones de partición del dominio y envio de mensajes entre subdominios, están escondidas de tal manera que el usuario no se distrae en estas tareas y se concentra en la solución numérica del problema.

El resultado de este ejemplo, en dos dimensiones, se muestra en la figura 6.16, donde se puede observar el campo de temperaturas calculado en 9 subdominios. La malla utilizada fue de 48×48.

El mismo problema se resolvió en tres dimensiones en una malla de $128 \times 128 \times$. Este ejemplo se utilizó para hacer un cálculo de la aceleración que se obtiene con diferente número de procesadores. El resultado de este experimento numérico se muestra en la tabla 6.5 y en la figura 6.17 se muestra la aceleración del código.

N. Proc.	Tiempo (segs.)	Aceleración	Eficiencia	Costo
1	9121.73	1.0	1.0	9121.73
2	4680.84	1.95	0.97	9361.68
4	2559.71	3.56	0.89	10238.84
6	1555.19	5.87	0.98	9331.14
8	1230.70	7.41	0.92	9845.60
9	1065.94	8.56	0.95	9593.46
12	873.16	10.45	0.87	10477.92

Cuadro 6.2: Métricas de rendimiento para la solución en paralelo del problema de convección natural forzada en 3D.

Se puede notar, en la tabla de arriba, que el valor de la eficiencia oscila en el valor de 0.9. Esto indica que el programa es lo suficientemente escalable aun con 12 procesadores.

En la gráfica de la figura 6.17 se observa que la aceleración es bastante buena, aún con 12 procesadores. Después de 12 no se tiene información pero se puede preveer que la aceleración se degrada y el costo empieza a subir considerablemente.

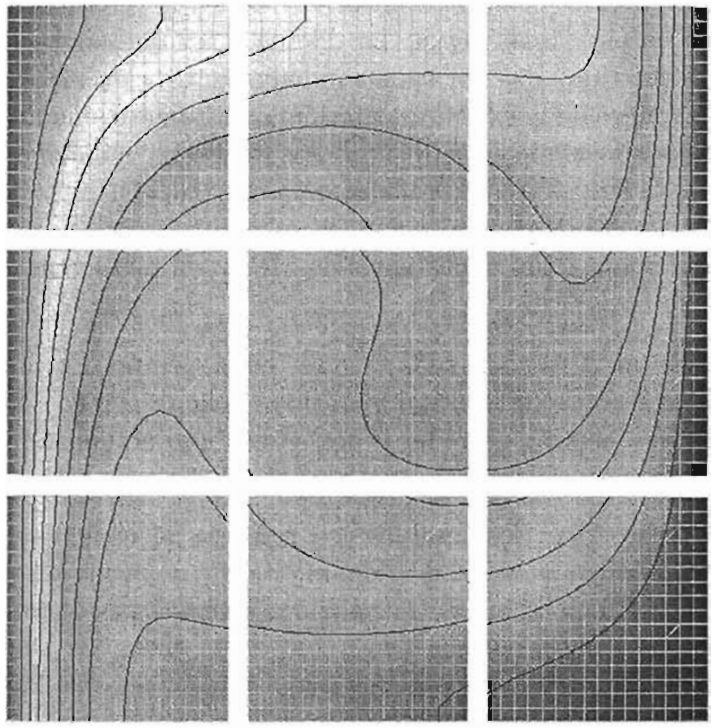


Figura 6.16: Resultado del cálculo en dos dimensiones para 9 procesadores.

Los resultados anteriores se obtuvieron en una computadora SGI ONIX 350 con 12 procesadores R16K. El modelo de memoria de esta computadora es de memoria compartida (NUMAFlex). Aunque el mismo código puede ejecutarse en un cluster de PC's en donde esté instalado MPI.

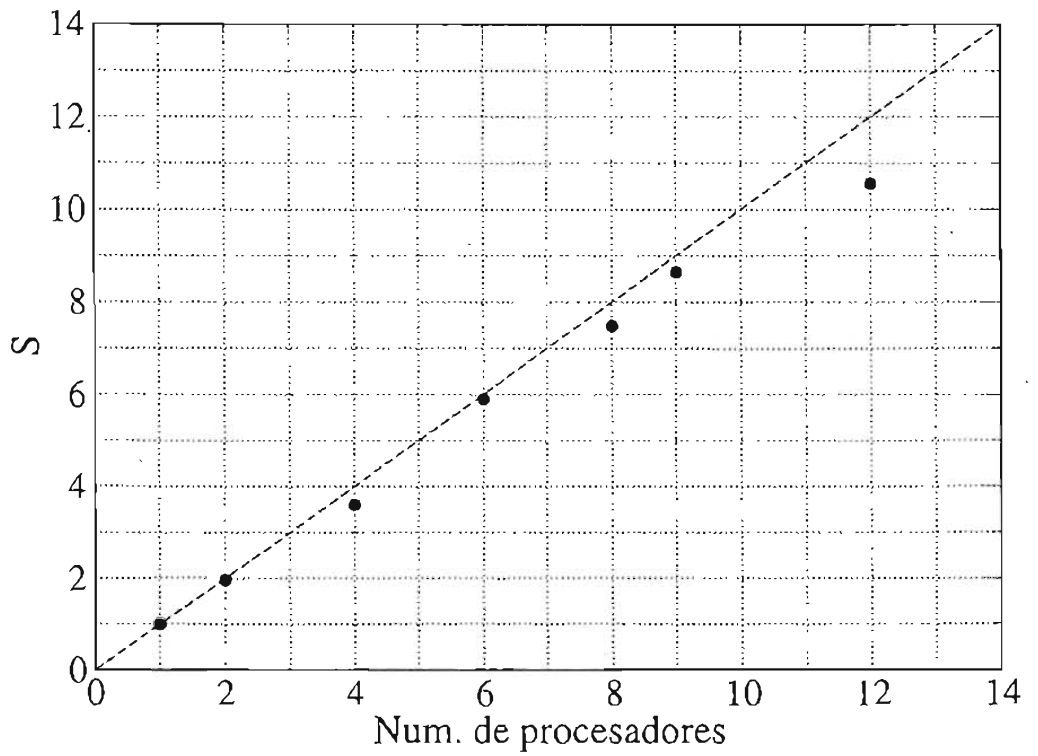


Figura 6.17: Aceleración del código para el problema tridimensional, resuelto en una ONIX 350 con 12 procesadores R16K.

Capítulo 7

Conclusiones finales

En este trabajo se construyó una biblioteca de clases y funciones para resolver problemas de convección en cavidades rectangulares. Solamente se consideraron fluidos newtonianos e incompresibles. Todos los ejemplos se resolvieron en mallas cartesianas uniformes. Se implementaron además tres esquemas diferentes para aproximar los términos convectivos y se incluyó un modelo de turbulencia basado en la LES. Finalmente se agregó un módulo para particionar el dominio y resolver algunos problemas en paralelo.

El sistema se desarrolló utilizando paradigmas de programación modernos, que generalmente no se toman en cuenta en la construcción de software para aplicaciones científicas. La programación orientada a objetos, la programación genérica y la programación en paralelo contienen herramientas que permiten desarrollar código ordenado, con un alto nivel de abstracción y eficiente. Estos beneficios fueron de utilidad durante el desarrollo y en la solución de los ejemplos mostrados en el capítulo anterior.

La construcción de sistemas donde se utiliza la POO se debe basar en un proceso bien ordenado de desarrollo de software. En este trabajo se concibió una metodología de desarrollo basada en la de Schimmel [65]. En nuestra metodología, además de incluir el modelo matemático y la discretización de las ecuaciones como etapas de desarrollo, como lo propone Schimmel, agregamos las etapas de generalización y la optimización, dado que nuestro objetivo era desarrollar un sistema muy general y eficiente. De esta manera se construyó el código en forma ordenada e incremental, lo cual permite que se puedan agregar otras componentes de manera sencilla. La arquitectura de desarrollo sobre la que basamos todo el sistema se deriva directamente de un esquema general de modelación computacional. Así, es muy claro, para un experto en modelación computacional, como y donde agregar módulos al sistema. Por ejemplo, si se desea resolver un problema en mallas no uniformes, se debe agregar una clase para este tipo de mallas, además debe implementar el esquema numérico que tome en cuenta la forma de estas mallas. Las otras partes del sistema no se modifican dado que son independientes, es decir se tiene bajo acoplamiento entre sus componentes.

En los ejemplos de uso mostrados en el capítulo anterior, se observa que uno de los beneficios de los paradigmas de programación utilizados, es que la definición de abstracciones complejas permite dejar de lado los detalles de su construcción y enfocar

la atención en como combinarlas para resolver un problema. Las abstracciones reflejan claramente la entidad del mundo real que representa, es decir se tiene alta cohesión.

Otro beneficio es que, la generalización de las ecuaciones permite que los códigos para resolver problemas distintos, consisten de aproximadamente el mismo número de pasos. El usuario de la biblioteca de clases aquí desarrollada, sólo tiene que conocer los pasos que se deben seguir para resolver un problema, conocer la forma en que se usan las clases (su interfaz) y combinarlas para que colaboren en la solución. Además, el conjunto base de clases es mínimo, por lo que el aprendizaje debe ser simple y rápido. Entonces, para usar la biblioteca, sólo es necesario aprender algunos comandos básicos y los pasos que se siguen para resolver un problema, de la misma manera en que se hace con software sofisticado como *Mathematica*[49] o *Matlab*[50]. Por supuesto que el sistema desarrollado en este trabajo no se compara con los antes mencionados, en el sentido que resuelve problemas desde otro punto de vista, y que aquellos son desarrollados por un equipo grande de personas y contienen muchas más facilidades y funciones.

Una vez que se conoce la base de la biblioteca, es posible agregar más clases para adicionar comportamientos nuevos al sistema. Debido a que las diferentes partes del sistema están divididas en módulos que se relacionan directamente con las partes de una modelación computacional típica, la extensión del sistema se puede hacer siguiendo la metodología propuesta en el capítulo 5.

Existen entonces dos niveles de uso de la biblioteca: el usuario, que utiliza las abstracciones y que no le interesa ni necesita conocer como están construidas internamente, y el programador de las abstracciones, que debe conocer la metodología de construcción y la forma de implementar las abstracciones.

Un ejemplo, en el que se realizó una extensión al sistema es el módulo de paralelismo. En la arquitectura inicial no se considerá el uso de múltiples procesadores para resolver un problema. Sin embargo, después fue posible agregar un conjunto de clases que hacen la descomposición del dominio y permiten ejecutar el programa en varios procesadores. Para aprovechar las clases construidas con las que se resuelven problemas en forma serial, se sigue una estrategia de descomposición de datos simple, como se explica en el capítulo 5, de tal manera que se resuelven subproblemas similares que se comunican para actualizar información y obtener una solución global consistente con la solución que se obtiene en un solo procesador. El resultado final es que el código en paralelo no cambia mucho con respecto del código serial para resolver un mismo problema.

Los resultados numéricos muestran un comportamiento cualitativo aceptable. En el ejemplo en dos dimensiones de la sección 6.2, se muestra además que, usando un esquema de alto orden (QUICK), es posible obtener resultados numéricos con muy buena precisión, como se observa en el ejemplo de la sección 6.2. En el ejemplo de flujo turbulento, también se obtuvieron resultados numéricos aceptables en comparación con resultados experimentales. Esto por un lado prueba que el modelo de turbulencia es confiable, y por el otro que el sistema resuelve correctamente las ecuaciones de balance. Finalmente, en el ejemplo de paralelismo se observa que la forma de los resultados no se modifica en comparación con el cálculo en un solo procesador, y además se obtiene una aceleración muy buena. Sin embargo, se debe tomar en cuenta que el problema

que se resuelve es lineal y esto puede ayudar a que los procesadores trabajen de forma equitativa (buen balance de carga).

En esta etapa del desarrollo se tiene una herramienta confiable, eficiente y de fácil uso, para resolver problemas de flujo convectivo en cavidades. Sin embargo, la extensión a problemas más complejos es directa. Por ejemplo, para resolver problemas con fronteras libres, la biblioteca lo puede hacer si se imponen las condiciones en dichas fronteras (flujo que entra y flujo que sale). Un ejemplo de esto se puede observar indirectamente en la solución de problemas en paralelo, en donde el dominio se rompe en varios subdominios y en la interfase entre ellos se imponen condiciones de frontera tipo Dirichlet, y esto no es otra cosa que la definición de subproblemas con frontera libre.

Si se desea resolver problemas en dominios irregulares, es posible construir mallas estructuradas curvilíneas que se acoplan al dominio irregular. Actualmente existen muchos algoritmos sofisticados para generar este tipo de mallas [74]. Estos algoritmos se pueden encapsular dentro de abstracciones que representen el tipo de malla que generen y adicionarlas a la jerarquía de herencia mostrada en la figura 5.5. Además, se deben agregar esquemas numéricos que utilicen estas mallas, de tal manera que las ecuaciones a resolver sean adaptadas por estos esquemas. Siguiendo los pasos de la metodología de desarrollo propuesta en esta tesis, será posible que las nuevas clases se incorporen de manera natural al sistema, permitiendo que la solución de problemas en dominios irregulares se lleve a cabo usando códigos similares a los mostrados en el capítulo 6.

La flexibilidad del sistema también puede aprovecharse para implementar otros modelos de turbulencia, y así poder comparar entre varios de ellos. Se ha visto en este trabajo que el sistema es confiable para resolver este tipo de flujos y que se se deben tomar en cuenta algunos criterios (como usar mallas no uniformes), para obtener buena precisión en los resultados numéricos.

El modelo de paralelismo usado en este trabajo ofrece una manera de reducir el tiempo de cálculo. Este modelo se conoce como paralelismo de grano grueso. En muchas aplicaciones numéricas se utiliza un paralelismo que se conoce como de grano fino, que se hace a nivel de las instrucciones del código y se implementa fácilmente en arquitecturas de memoria compartida. Este último modelo ha mostrado una ganancia muy buena en diferentes tipos de aplicaciones. Aunque en este trabajo no se utilizó el paralelismo de grano fino, la extensión del sistema para agregar esta característica se puede hacer directamente. Gracias al bajo acoplamiento de las partes del sistema, es posible agregar algoritmos paralelos para resolver los sistemas lineales de ecuaciones. Estos algoritmos son de grano fino. Así, se puede combinar paralelismo de grano grueso, descomposición de dominio, con el paralelismo de grano fino, algoritmos paralelos para los sistemas lineales. Esta combinación se conoce como paralelismo de varios niveles y puede ser útil en la nueva técnología que actualemente está de moda y que se conoce como GRIDs [22]. La idea de los GRIDs es construir organizaciones virtuales con miembros institucionales que posean equipos de supercómputo. En esta organización virtual, cada participante otorga permisos a usuarios de todas las instituciones para usar sus equipos y de esta manera resolver problemas de gran escala que no pueden ser resueltos con un sólo equipo.

La conclusión final es que, aunque no se ha resuelto el problema de la crisis del

software en la construcción de sistemas para aplicaciones científicas, si se ha reducido considerablemente la "entropía" del software, es decir, el sistema aquí construido está bien ordenado, es de fácil uso y extensión, y es capaz de resolver problemas complejos de la misma área, obteniendo soluciones físicamente realistas y con buena precisión si se toman encuenta algunos criterios.

7.1. Recomendaciones

Existen algunas recomendaciones útiles para decidir entre utilizar o no programación orientada a objetos, programación genérica y programación en paralelo en la solución de problemas de cómputo científico. Estas recomendaciones son las siguientes:

1. Utilizar una biblioteca ya construida que incluya los paradigmas antes mencionados, y utilizar sus abstracciones sin poner atención en como están construidas. En este caso se debe tener la confianza de que dicha biblioteca entrega soluciones adecuadas.

2. Desarrollar la biblioteca. En este caso, los requerimientos que se deben cumplir son:

   a) Disponibilidad. Es decir se debe contar con plataformas donde probar el código, compiladores sofisticados, y mucha bibliografía.

   b) Necesidad. Es decir, se debe justificar la necesidad de construir una nueva biblioteca que beneficie a un número grande de personas.

   c) Soporte. Es decir, que haya suficiente información de las plataformas y los paradigmas que se van a utilizar y que las personas que los desarrollarán puedan entrenarse en esos tópicos.

   d) Continuidad y documentación. Es decir, las personas que inicien el proyecto deben mantenerse por un largo tiempo en el desarrollo. Además se debe documentar cada parte del proyecto para que no se pierda ningún detalle en caso de que algún integrante deje el desarrollo.

   e) Competencia. El equipo encargado del desarrollo debe ser experto en una cierta área y conocer un poco de las otras áreas del conocimiento que estén involucradas en el desarrollo del proyecto.

   f) Tiempo. Es decir, dedicar un tiempo considerable a las actividades del desarrollo.

   Si alguna de los requerimientos antes expuestos no se cumple, es mejor elegir la opción 1.

Dadas estas recomendaciones, se tienen varias opciones para lograr un desarrollo de software de calidad. Si se cuenta con un equipo de personas grande con experiencia en

programación y en la teoría de los problemas que se van a resolver, entoces es adecuado iniciar un desarrollo propio. En otro caso, posiblemente lo mejor sea involocrase en un desarrollo en donde el código sea de dominio público y permita hacer adiciones a cada usuario. Este último caso es que prentedemos seguir para que el sistema de este trabajo se enriquezca con la experiencia de otros investigadores y programadores.

7.2. Trabajo futuro

El sistema desarrollado en este trabajo permite resolver una cantidad considerable de problemas de convección natural en cavidades rectangulares. Debido a la forma en que se construyó este software, es posible adicionar nuevas características de forma simple. Algunas de las características que se pueden agregar en corto plazo son:

- Manejo de mallas estructuradas no uniformes.
- Estudiar algoritmos alternos al SIMPLEC (como el CLEAR).
- Implementación de otros algoritmos más eficiente de solución del sistema lineal de ecuaciones (Multigrid, SIP, etc.).
- Otros esquemas numéricos tanto para los términos difusivos como para los convectivos.

A mediano y largo plazo se espera poder incluir lo siguiente:

- Algoritmos paralelos de solución de los sistemas lineales.
- Otros modelos para simular la turbulencia.
- Manejo de dominios irregulares a través de mallas estructuradas curvilíneas.
- Solución de otro tipo de problemas.

Todo lo anterior es posible incluirlo de forma ordenada si se sigue el proceso de desarrollo discutido en el capítulo 5 y se toma en cuenta la arquitectura del sistema actual.

Bibliografía

[1] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison–Wesley, 1999.

[2] R. Bastiaans, C. Rindt, F.Ñieuwstadt, and A. van Steenhoven. Direct and large–eddy simulation of the transitional of two– and three–dimensional plane plumes in a confined enclosure. *Int. J. Heat and Mass Trans*, 43:2375–2393, 2000.

[3] G. K. Batchelor. *An Introduction to Fluid Dynamics.* Cambridge University Press, 1970.

[4] K. Beck. *Extreme programming explained : embrace change.* Addison–Wesley, 2000.

[5] Blas (basic linear algebra subprograms). http://www.netlib.org/blas/.

[6] Blitz++: Arrays for object oriented scientific computing. http://www.oonumerics.org/blitz/.

[7] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin–Cummings, Redwood City, CA, 1994.

[8] D. Bulka and D. Mayhew. *Efficient C++: Performance Programming Techniques.* Addison Wesley Longman, 2000.

[9] A. Burks, H. Goldstine, and J. von Neumann. *Preliminary Discussion of the Logical Design of an Electronical Computing Instrument*, volume 5 of *In: Taub, A.H. (Editor), Collected Works of John von Neumann.* New York, Macmillan, 1963, 1946.

[10] M. Carroll and M. Ellis. *Designing and Coding Reusable C++.* Addison–Wesley, Reading, MA, 1995.

[11] T. F. Chan and T. P. Mathew. *Domain Decomposition algorithms.* Acta Numerica, Vol. 1, pp. 61 – 141, 1994.

[12] S. Chandrasekhar. *Hydrodinamic and Hidromagnetic Stability.* Clarendon Press, 1961.

[13] L. de la Cruz, I. García, V. Goday, and E. Ramos. Parallel lagrangian visualization applied to natural convection flows. In *Symposium on Parallel and Large Data Visualization and Graphics*, IEEE Visualization 2001. IEEE, 2001.

[14] G. de Vahl Davis. Natural convection of air in a square cavity: a bench mark numerical solution. *Int J Numer Methods Fluids*, 3:249–264, 1983.

[15] C. R. Doering and D. J. Gibbon. *Applied analysis of the Navier-Stokes equations.* Cambridge University Press, 1995.

[16] J. V. Doormal and G. D. Raithby. Enhancements of the simple method for predicting incompressible fuiod flows. *Num. Heat Transfer*, 7:147–163, 1984.

[17] T. Eidson. Numerical simulation of the turbulent rayleigh–bénard problem using subgrid modelling. *J. Fluid Mech*, 158:245–268, 1985.

[18] J. H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics.* Springer–Verlag, second edition edition, 1999.

[19] C. Fletcher. *Computational Techniques for Fluid Dynamics 1: Fundamentals and General Techniques.* Springer–Verlag, second edition edition, 1991.

[20] C. Fletcher. *Computational Techniques for Fluid Dynamics 1: Specific Techniques for Different Flow Categories.* Springer–Verlag, second edition edition, 1991.

[21] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[22] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Intl. Jour. Supercomputer App.*, 15(3), 2001.

[23] B. Galperin and S. A. Orzag. *Large eddy simulation of complex engineering and geophysical flows.* Cambridge University Press, 1993.

[24] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B.Ñitzberg, W. Sapir, and M. Snir. *MPI: The Complete Reference: Volume 2, The MPI-2 Extensions.* The MIT Press, Cambridge, Massachusetts, London, second edition edition, 1998.

[25] T. Han, J. Humphrey, and B. E. Launder. A comparison of hybrid and quadratic-upstream differencing in high reynolds number elliptic flows. *Comp. Meth. in App. Mech. and Eng.*, 29:81–95, 1981.

[26] T. Hayase, J. Humphrey, and R. Greif. A consistently formulated quick scheme for fast and stable convergence using finite-volume iterative calculation procedures. *J. of Comp. Phys*, 98:108–118, 1992.

[27] J. R. Herring, D. Schertzer, M. Lesieur, G. R. Newman, J. P. Chollet, and M. Larcheveque. A comparative assessment of spectral closures as applied to passive scalar diffusion. *J. Fluid Mech.*, 124:411–437, 1982.

[28] C. Hirsch. *Numerical Computation of Internal and External Flows Volume 1: Fundamentals of Numerical Discretization.* John Wiley & Sons, 1988.

[29] C. Hirsch. *Numerical Computation of Internal and External Flows Volume 2: Computational Methods for Inviscid and Viscous Flows.* John Wiley & Sons, 1988.

[30] High performace fortran. http://www.crpc.rice.edu/HPFF/.

[31] K. Hsieh and F. Lien. Numerical modelling of buoyancy-driven turbulent flows in enclosures. *Int. J. Heat and Fluid*, 25:659 – 670, 2004.

[32] P. G. Huang, B. E. Launder, and M. A. Leschziner. Discretization of non-linear convection processes: A broad-range comparison of four schemes. *Comp. Meth. in App. Mech. and Eng.*, 48:1–24, 1985.

[33] Internet 2, universidad nacional autónoma de méxico. http://www.internet2.unam.mx/.

[34] Imsl mathematical and statistical libraries. http://www.vni.com/products/imsl/.

[35] Intel homepage:. http://www.intel.com/technology/silicon/mooreslaw/.

[36] Y. Ishikawa, R. Oldehoeft, J. Reynders, and M. Tholburn, editors. *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lectures Notes in Computers Science.* Springer, 1997.

[37] I. Jacobson. *Object-oriented software engineering : A use case driven approach.* Addison–Wesley, 1992.

[38] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison–Wesley, 1999.

[39] D. Jang, R. Jetli, and S. Acharya. Comparison of the piso, simpler and simplec algorithms for the treatment of pressure-velocity coupling in steady flow problems. *Numer. Heat Transfer*, 19:209 – 228, 1986.

[40] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison–Wesley, 1999.

[41] J. Kim, P. Moin, and R. Moser. Turbulence statistics in fully developed chanel flow at low reynolds number. *J. of Fluid Mech.*, 177:133–166, 1987.

[42] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin–Cummings, Redwood City, CA, 1994.

[43] O. A. Ladyeskaya. *The matemathical theory of viscous incompressible flow.* Gordon and Breach, New York, 1963.

[44] Lapack – linear algebra package. http://www.netlib.org/lapack/.

[45] C. Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design.* Prentice–Hall, 1998.

[46] M. A. Leischziner. Practical evaluation of three finite difference schemes for computation of steady-state recirculating flows. *Comp. Meth. in App. Mech. and Eng.*, 293:293–312, 1980.

[47] B. P. Leonard. A stable and accurate conevctive modelling procedure based on quadratic upstream interpolation. *Comp. Meth. in App. Mech. and Engineering*, 19:59–98, 1979.

[48] M. Lesieur and O. Métais. New trends in large-eddy simulations of turbulence. *Annu. Rev. Fluid Mech.*, 28:45–82, 1996.

[49] Mathematica home page. http://www.wolfram.com/.

[50] Matworks home page. http://www.mathworks.com/.

[51] P. Moin and J. Kim. Tackling turbulence with supercomputers. *Scientific American*, 1, 1997.

[52] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[53] Multilevel parallel programming. http://foxtrot.ncsa.uiuc.edu:8900/public/MULTI/.

[54] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide.* Addison–Wesley, 2001.

[55] Nag : Numerical algorithms group. http://www.nag.co.uk/.

[56] Openmp : Simple, portable, scalable smp programming. http://www.openmp.org/.

[57] The object-oriented numerics page. http://www.oonumerics.org/.

[58] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow.* McGraw–Hill, 1980.

[59] S.-H. Peng and L. Davidson. Large eddy simulation for turbulent buoyant flow in a confined cavity. *Int. J. Heat and Fluid*, 22:323–331, 2001.

[60] A. Pollard and A. L. Siu. The calcualtion of some laminar flows using various discretization schemes. *Comp. Meth. in App. Mech. and Eng.*, 35:293–313, 1982.

[61] Parallel virtual machine. http://www.csm.ornl.gov/pvm/pvm_home.html.

[62] R. S. Rogallo and P. Moin. Numerical simulation of turbulent flows. *Annu. Rev. Fluid Mech.*, 16:99–137, 1984.

[63] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design.* Prentice–Hall, 1991.

[64] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Boston: PWS Publishing Company, 2000.

[65] F. Schimmel. An object-oriented design for and atmospheric flow model. In *Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing, POOSC 03.* Forschungszentrum Julich, Zentralinstitut fur Angewandte Mathematik, 2003.

[66] J. Smagorinsky. General circulation experiments with the primitive equations. Part I: The basic experiment. *Mon. Wea. Rev.*, 91(99 - 164), 1963.

[67] B. F. Smith, P. E. Bjorstad, and W. D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.

[68] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference: Volume 1, The MPI Core.* The MIT Press, Cambridge, Massachusetts, London, second edition edition, 1998.

[69] T. Stevens and R. Pooley. *Using UML Software engineering with objects and components.* Addison–Wesley, 2000.

[70] B. Stroustrup. *The C++ Programming Language.* Addison–Wesley, third edition edition, 1997.

[71] W. Q. Tao, Z. G. Qu, and Y. L. He. A novel segregated algorithm for incompressible fluid flow and heat transfer problems – clear (coupled and linked equations algorithm revised) part i: Mathematical formulation and solution procedure. *Num. Heat Transfer*, 45:1–17, 2004.

[72] H. Tennekes and J. L. Lumley. *A Firts Course in Turbulence.* MIT Press Cambdrige, 1972.

[73] Thomas B. Gatski (NASA Langley Research Center), M. Youssuff Hussaini (ICASE), and Jonh L. Lumley (Cornell University), editors. *Simulation and Modeling of Turbulent Flows*, New York, 1996. Oxford University Press.

[74] J. F. Thompson, Z. U. A. Warsi, and C. W. Mastin. *Numerical Grid Generation and Applications.* North-Holland, 1985.

[75] Y. Tian and T. Karayiannis. Low turbulence natural convection in an air filled square cavity, part i: The thermal and fluid flow fields. *International Journal of Heat and Mass Transfer*, 43:849 - 866, 2000.

[76] Y. Tian and T. Karayiannis. Low turbulence natural convection in an air filled square cavity, part ii: The thermal and fluid flow fields. *International Journal of Heat and Mass Transfer*, 43:867 - 884, 2000.

[77] Top 500 supercomputers site. http://www.top500.org/.

[78] L.Ñ. Trefethen. *Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations.* unpublished text, disponible en http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/pdetext.html, 1996.

[79] ublas, a c++ template class library that provides blas level 1, 2, 3 functionality for dense, packed and sparse matrices. http://www.boost.org/libs/numeric/ublas/doc/index.htm.

[80] E. Unruh. Prime number computation, 1994. ANSI X3J16-94-0075/ISO WG21-462.

[81] D. Vandevoorde and N. M. JosuttisOB. *C++ Templates.* Addison–Wesley, 2003.

[82] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[83] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[84] H. Versteeg and W. Malalasekera. *An introduction to computational fluid dynamics: The finite volume method.* Longman, 1995.

[85] Xprogramming an extreme programming resource. http://www.extremeprogramming.org/.

[86] Extreme programming: A gentle introduction. http://www.xprogramming.com/.

Apéndice A

Biblioteca de clases

A.1. Clases principales

Mesh<T_mesh> : Clase base abstracta polimórfica que encapsula las características generales de todo tipo de mallas. Los objetos que representan mallas de un tipo particular se crean a partir de clases especializadas que se derivan de esta clase base. En esta clase se definen algunos algoritmos generales que se adaptan de acuerdo con el tipo de malla mediante el parámetro T_mesh.

StructuredMesh<T_number, int Dim> : Esta es una clase particular que define a las mallas de tipo estructuradas uniformes. Se deriva de la clase Mesh y se utilizan dos parámetros que son, el tipo de número para calcular los puntos (`float` o `double`) y la dimensión de la malla `Dim`. Se utiliza herencia parametrizada recursiva (Barton & Nackman), véase figura A.1.

DiagonalMatrix<T_matrix> : Clase para matrices diagonales cuyo objetivo principal es almacenar los datos de los sistemas lineales descritos en la sección 4.5. Esta es una clase polimórfica cuyo parámetro es el tipo de matriz. Los adaptadores que acepta esta clase son Tri, Penta y Hepta para matrices tridiagonales, pentadiagonales y heptadiagonales respectivamente.

GeneralEquation<T_number, int Dim> : En esta clase se combinan dos conceptos: ecuación general (3.1) y su forma discreta, ecuación (4.6). Aquí y se concentran las características generales de todas las ecuaciones. Dos parámetros son utilizados para definir la presición (`float` o `double`) y la dimensión del problema. De esta clase se derivan las ecuaciones particulares definidas en las secciones 3.1 y 3.2.

EnergyEquation<T_number, int Dim, T_approx> : Clase que representa el concepto de la ecuación de energía, ecuación (3.14). Se deriva de la clase general GeneralEquation, de tal manera que tiene acceso a todas las características definidas en la clase base. Para tomar en cuenta los diferentes esquemas numéricos, se utiliza herencia parametrizada recursiva y de esta manera, se derivan subclases en las que

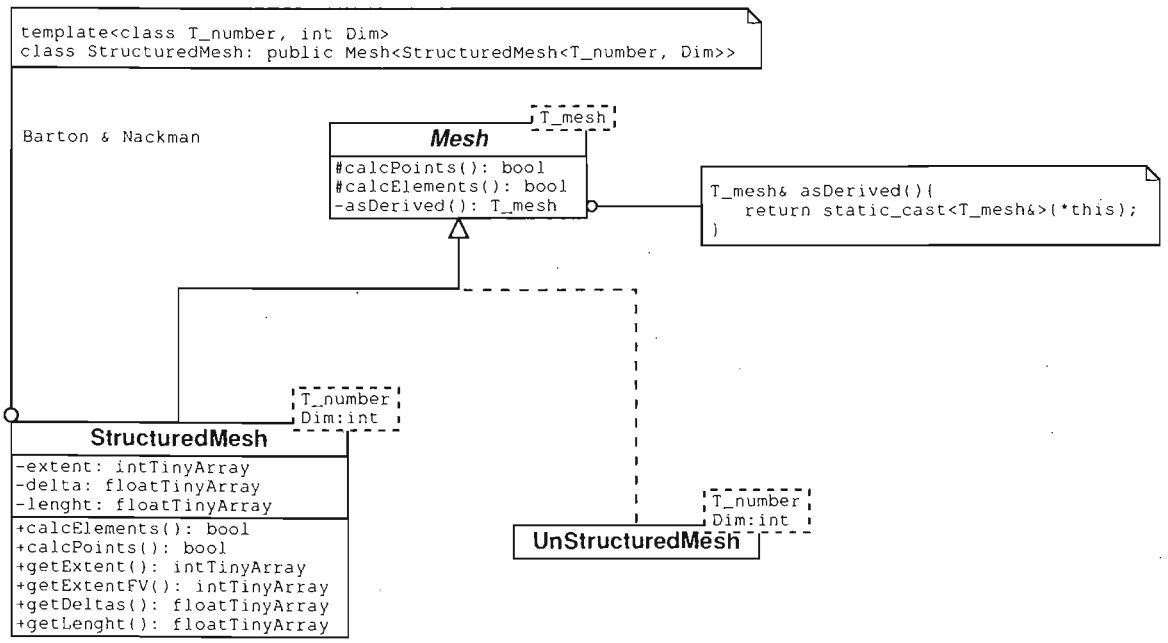


Figura A.1: Jerarquía de clases

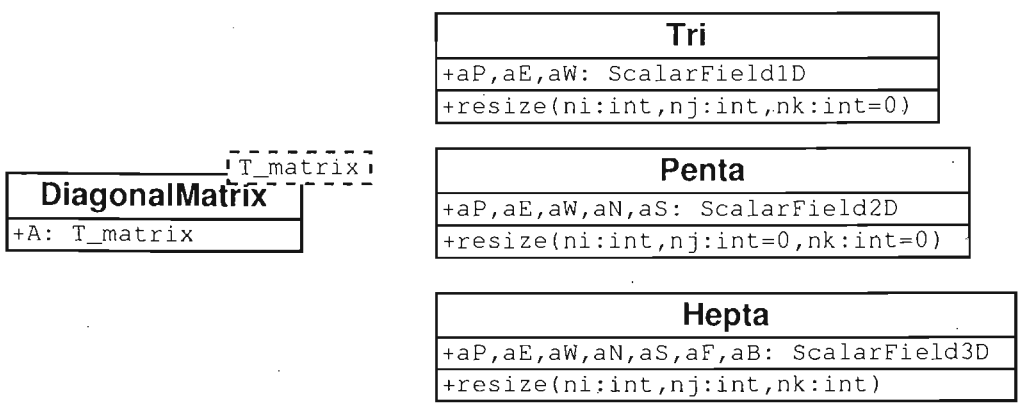


Figura A.2: Diagrama de clases: matrices.

se implementan dichos esquemas. Estas subclases funcionan como adaptadores y parametrizan a la clase base mediante el parámetro T_approx. De esta forma se adaptan los algoritmos que calculan los coeficientes del sistema lineal. Por tanto, esta es una clase polimórfica y cada adaptador requiere lo siguiente:

- Implementación del método `calcCoefficients()` en donde se debe definir como se calculan los coeficientes (difusivos y convectivos). Esto depende del esquema numérico que se desee utilizar.

- El método anterior debe almacenar los coeficientes en los arreglos aP,aE, aW, aN, aS, aF, aB y sp, definidos en la clase base GeneralEquation. Estos arreglos corresponden a los definidos en el sistema lineal general de la ecuación (4.6).
- Los cálculos que se realicen sobre los arreglos antes mencionados, se deberán hacer dentro de los límites: bi = inicio del arreglo en dirección i; ei = fin del arreglo en dirección i; y los correspondientes para j y k. Estos números se definen en la clase GeneralEquation.
- La aplicación de las condiciones de frontera se debe hacer mediante el método applyBoundaryConditions() de GeneralEquation.

[XYZ]MomentumEquation<T_number, int Dim, T_approx> : Clases que representan el concepto de las ecuaciones de cantidad de movimiento, ecuación (3.13). Se derivan de la clase general GeneralEquation, de tal manera que tiene acceso a todas las características definidas en la clase base. Estas son clases polimórficas que se adaptan a través del parámetro T_approx y los requerimientos para los adaptadores son similares a los de la clase EnergyEquation.

PressureEquation<T_number, int Dim> : Clase que representa el concepto de la ecuación de presión (o corrección a la presión) que aparece en los métodos de tipo SIMPLE. Se deriva de la clase general GeneralEquation, de tal manera que tiene acceso a todas las características definidas en la clase base. A diferencia de las clases anteriores, esta no es una clase polimórfica dado que los coeficientes se calculan a partir de los coeficientes de las ecuaciones de cantidad de movimiento, de tal manera que se adapta indirectamente a través de la adaptación de las clases de cantidad de movimiento.

A.1.1. Clases para descomposición de dominio

Las siguientes tres clases encapsulan las operaciones para realizar la descomposición de dominio para ejecutar el programa en paralelo. Se ha utilizado un modelo de programación para arquitecturas de tipo SIMD, es decir, se ejecuta el mismo programa en cada procesador con diferentes datos. La comunicación se realiza mediante envío de mensajes usando llamadas a funciones de MPI.

Domain<T_mesh, int Dim> : Las características del dominio global donde se resuelve el problema, se encapsulan en esta clase. El parámetro T_mesh permite que el dominio sea discretizado, en principio, con cualquier tipo de malla. En este trabajo sólo se utilizan mallas estructuradas uniformes. El dominio se descompone en varios subdominios, por lo que esta clase contiene como atributos objetos de la clase Subdomain. La descomposición se realiza mediante el método decomposition() el cual está parametrizado con el tipo de partición a realizar. La partición es una clase que encapsula las operaciones que se deben realizar para que cada subdomino esté bien definido (número de puntos, rangos, etc.).

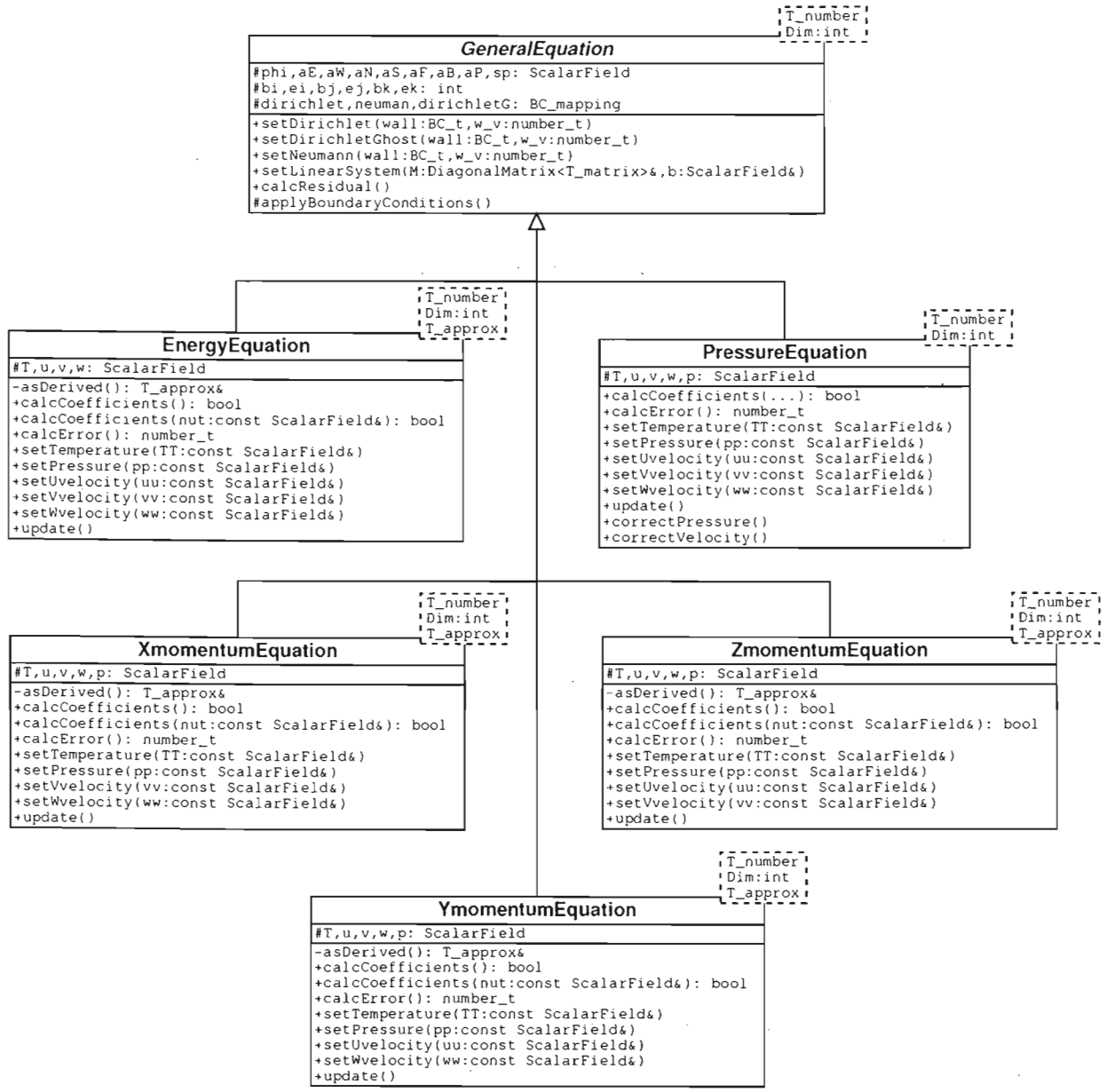


Figura A.3: Diagrama de clases: ecuaciones.

SubDomain<T_mesh, int Dim> : Esta clase se encarga de manejar los subdominios que resultan de la descomposición de dominio. Está parametrizada con los mismos parámetros que la clase Domain, de tal manera que todos los subdominios tienen las mismas características del dominio global. Cuando se realiza la partición del dominio global, se construye una topología virtual de tipo cartesiana como la

mostrada en la figura 5.10. En la inicialización de cada subdominio se realiza lo siguiente:

- Se calcula la posición de cada subdominio dentro de la topología virtual.
- Se determinan los subdominios vecinos y se inicializan los sockets de comunicación entre ellos.
- Se determina la información que será intercambiada

Lo anterior se realiza una sola vez y esa información persiste durante todo el cálculo. Esto significa que el estado de los objetos de tipo subdominio permanece sin cambio. La topología virtual se realiza mediante el uso de la función :

```
Cartcomm Intracomm::Create_cart(int ndims, const int dims[],
                                const bool periods[], bool reorder) const
```

que genera una topología cartesiana para ordenar los procesadores, donde `ndims` indica el número de dimensiones de la topología, `dims` es un arreglo en donde se especifica el número de bloques que se tendrá en cada dirección, `periods` determina si la malla es periódica y `reorder` es una variable boolena que permite ordenar los procesadores de acuerdo a la topología física del equipo donde se ejecute.

Una vez construida la topología, el intercambio de información se realiza de la sigiente manera:

- Primero se determina la información que se va intercambiar entre cada subdominio mediante la función `Create_vector()` la cual construye un vector de datos para ser enviados entre los procesadores.
- Después, se asigna cada vector a las funciones que hacen el envío y recepción de información. Dichas funciones son `Send_init()` y `Recv_init()` respectivamente. Estas son funciones persistentes que incializan los *sockets* entre los procesadores. Dado que se conocen desde un inicio que subdominios deben intercambiar información, estas funciones son ideales para crear los puentes de comunicación desde un inicio y no hacerlo cada vez que se requiera enviar información.
- En el momento en que los subdominios han resuelto sus problemas locales, se puede empezar a realizar el intercambio de información. En este caso sólo es necesario aplicar la función `Start()` a los objetos generados por las funciones `Send_init()` y `Recv_init()`. De esta manera se inicia el intercambio de información.

Adicionalmente, en todos los subdominios se requiere la definición de condiciones de frontera reales y fantasma. Cada subdominio puede determinar, a partir de su posición dentro de la topología virtual, el lugar donde necesita condiciones de frontera reales y fantasma.

Blocks<int Dim> : Esta clase permite realizar particiones en dominios discretizados con mallas estructuradas uniformes. Recibe como argumento el número de bloques que se desean en cada direcciíon. Determina principalmente el número de puntos en cada subdominio y los rangos locales para cada campo escalar.

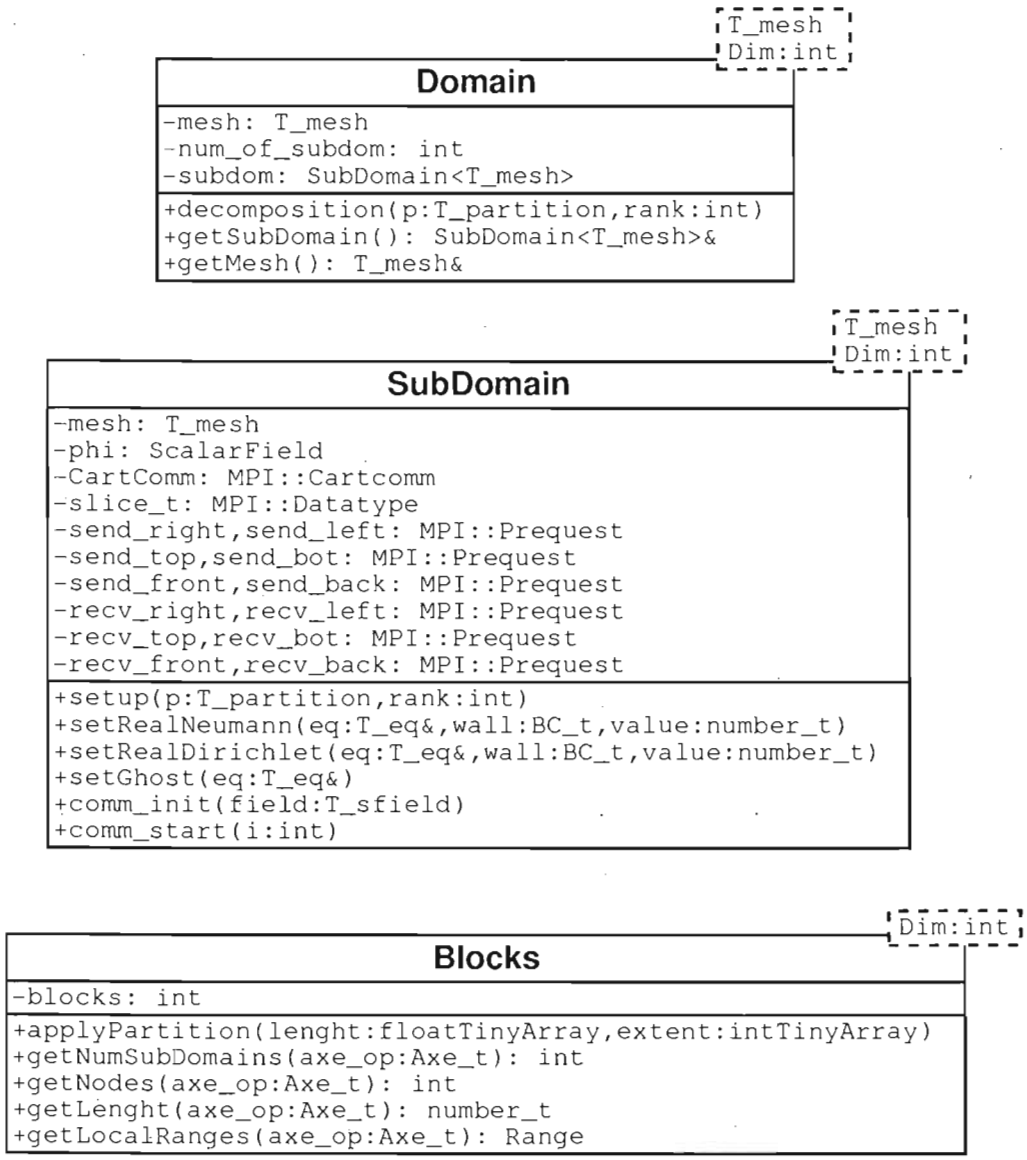


Figura A.4: Diagrama de clases: Descomposicoón de dominio.

A.1.2. Adaptadores

Los adaptadores son clases que permiten a los datos y métodos de otras clases modificar su estado y comportamiento. La adaptación se realiza en tiempo de compilación y por esta razón se conoce a esta característica de adaptación como polimorfismo estático. A diferencia del polimorfismo dinámico, que se realiza mediante el uso de funciones virtuales, el polimorfismo estático no introduce factores que afecten el rendimiento del código.

Los adaptadores que se implementaron en este trabajo son los siguientes:

Tri : Clase que adapta la clase DiagonalMatrix y la hace funcionar como una matriz de tres diagonales como la que se muestra en la figura 4.9.

Penta : Clase que adapta la clase DiagonalMatrix y la hace funcionar como una matriz de cinco diagonales como la que se muestra en la figura 4.10.

Hepta : Clase que adapta la clase DiagonalMatrix y la hace funcionar como una matriz de siete diagonales como la que se muestra en la figura 4.11.

Laplace<T_number, int Dim> : Clase que adapta a EnergyEquation para hacerla funcionar como la versión discreta de la ecuación de Laplace. Define un esquema lineal para aproximar los términos difusivos.

Diffusion<T_number, int Dim> : Clase que adapta a EnergyEquation para hacerla funcionar como la versión discreta de la ecuación de difusión. Define un esquema lineal para aproximar los términos difusivos.

UpwindE<T_number, int Dim> : Clase que adapta a EnergyEquation e implementa los coeficientes de la versión ecuación de energía 3.14, usando el esquema upwind descrito en la sección 4.3.1.

UpwindX<T_number, int Dim> : Clase que adapta a XmomentumEquation e implementa los coeficientes de la versión discreta ecuación de cantidad de movimiento en dirección x, usando el esquema upwind descrito en la sección 4.3.1. Los clases **UpwindY<T_number, int Dim>** y **UpwindZ<T_number, int Dim>** realizan una adaptación similar con las clases YmomentumEquation y ZmomentumEquation respectivamente.

CentralDiffE<T_number, int Dim> : Clase que adapta a EnergyEquation e implementa los coeficientes de la versión ecuación de energía 3.14, usando un esquema de diferencias centrales como el descrito en la sección 4.3.2.

CentralDiffX<T_number, int Dim> : Clase que adapta a XmomentumEquation e implementa los coeficientes de la versión discreta de la ecuación de cantidad de movimiento en dirección x, usando un esquema de diferencias centrales como el descrito en la sección 4.3.2. Los clases **CentralDiffY<T_number, int Dim>**

y **CentralDiffZ<T_number, int Dim>** realizan una adaptación similar con las clases YmomentumEquation y ZmomentumEquation respectivamente.

QuickE<T_number, int Dim> : Clase que adapta a EnergyEquation e implementa los coeficientes de la versión ecuación de energía 3.14, usando el esquema QUICK descrito en la sección 4.3.3.

QuickX<T_number, int Dim> : Clase que adapta a XmomentumEquation e implementa los coeficientes de la versión discreta de la ecuación de cantidad de movimiento en dirección x, usando el esquema QUICK descrito en la sección 4.3.3. Los clases **QuickY<T_number, int Dim>** y **QuickZ<T_number, int Dim>** realizan una adaptación similar con las clases YmomentumEquation y ZmomentumEquation respectivamente.

A.2. Espacios de nombres

Los espacios de nombres son componentes del sistema que contienen funciones y datos, pero a diferencia de las clases estos sólo existen una vez durante la ejecuación del programa, es decir, no se pueden instanciar objetos. Los espacios de nombres son similares a los módulos de Fortran 90. En esta implementación son útiles para agrupar algoritmos genéricos que tienen características en común, pero que semánticamente no pueden incluirse en ninguna clase.

A.2.1. Solver

En este espacio de nombres se agrupan los algoritmos para la solución de los sistemas lineales. Los algoritmos que contiene son:

thomas<Vector> : Algorithmo TDMA[1] descrito en la sección 4.5.1.

SolveByLines<T_eq> : Esta función implementa un algoritmo para resolver los sistemas lineales recorriendo la malla por líneas, como se describio en la sección 4.5. Combina diferentes direcciones de barrido para reducir el residuo del sistema. Se parametriza con el tipo de ecuación para que trabaje indistintamente con todas las ecuaciones.

line[XYZ]_[23]D<T_eq> : Funciones que implementan el algoritmo TDMA en las tres diferentes direcciones para dos y tres dimensiones. Se parametrizan con el tipo de ecuación para que trabajen indistintamente con todas las ecuaciones.

[1]También conocido como algoritmo de Thomas

A.2.2. LES_SSF

En este espacio de nombres se implementan los esquemas numéricos para la simulación de vórtices grandes. En este caso se implementan adaptadores similares a Upwind[EXYZ], CentralDiff[EXYZ] y Quick[EXYZ], en donde se toma en cuenta los términos que provienen del modelo submalla descrito en la sección 3.2. Estos adaptadores se utilizan de la misma forma en que se usan los correspondientes para el flujo laminar, véase sección B.2.3.

Además, en este espacio de nombres se incluyen las siguientes dos funciones:

calcTurbulentViscosity<T_sfield> : Función para calcular la viscosidad turbulenta a partir de la velocidad.

selectTurbulentViscosity<T_sfield> : Función para seleccionar los lugares donde la viscosidad turbulenta es diferente de cero. Este cálculo depende de la vorticidad.

A.2.3. NumUtils

Espacio de nombres con algunas funciones que implementan algunos algoritmos numéricos simples.

stagger[XYZ]<T_field> : Realiza interpolaciones de la malla original a las diferentes mallas desplazadas en las direcciones [XYZ].

inter[XYZ]<T_field> : Realiza la operaciones contraria de la función stagger[XYZ].

A.2.4. DiffOperators

Este espacio de nombres tiene como propósito incluir varios operadores numéricos. Actualmente sólo tiene uno que es útil para la turbulencia:

vorticity<T_number, int Dim> : Calcula la vorticidad del flujo.

A.3. Traits

Los Traits que hemos implementado, permiten intercambiar entre diferentes bibliotecas para manejar los arreglos. En este trabajo se utilizó la biblioteca Blitz++.

container_trait<T_number, int Dim> : Mapea el tipo de arreglo de una biblioteca particular al nombre container_trait. De esta manera, en el desarrollo de la biblioteca se utiliza este último nombre para definir arreglos.

tiny_container_trait<T_number, int Dim> : Mapea el tipo de arreglo de una biblioteca particular al nombre tiny_container_trait. Estos arreglos son de extensión limitada (a lo más 10), y en este trabajo se utilizan para definir las extensiones de otros arreglos en la diferentes direcciones.

A.3.1. Output

Espacio de nombres donde se agrupan diferentes funciones que permiten leer y escribir la solución en diferentes archivos.

printToFile_GP<T_sfield> : Imprime la solución en un formato simple para graficarlo con GNUPlot.

printToFile_DX<T_sfield> : Imprime la solución en un formato para graficarlo con OpenDX.

readFromFile_DX<T_sfield> : Lee la solución guardada con `printToFile_DX` de un determinado archivo y la coloca en un arreglo.

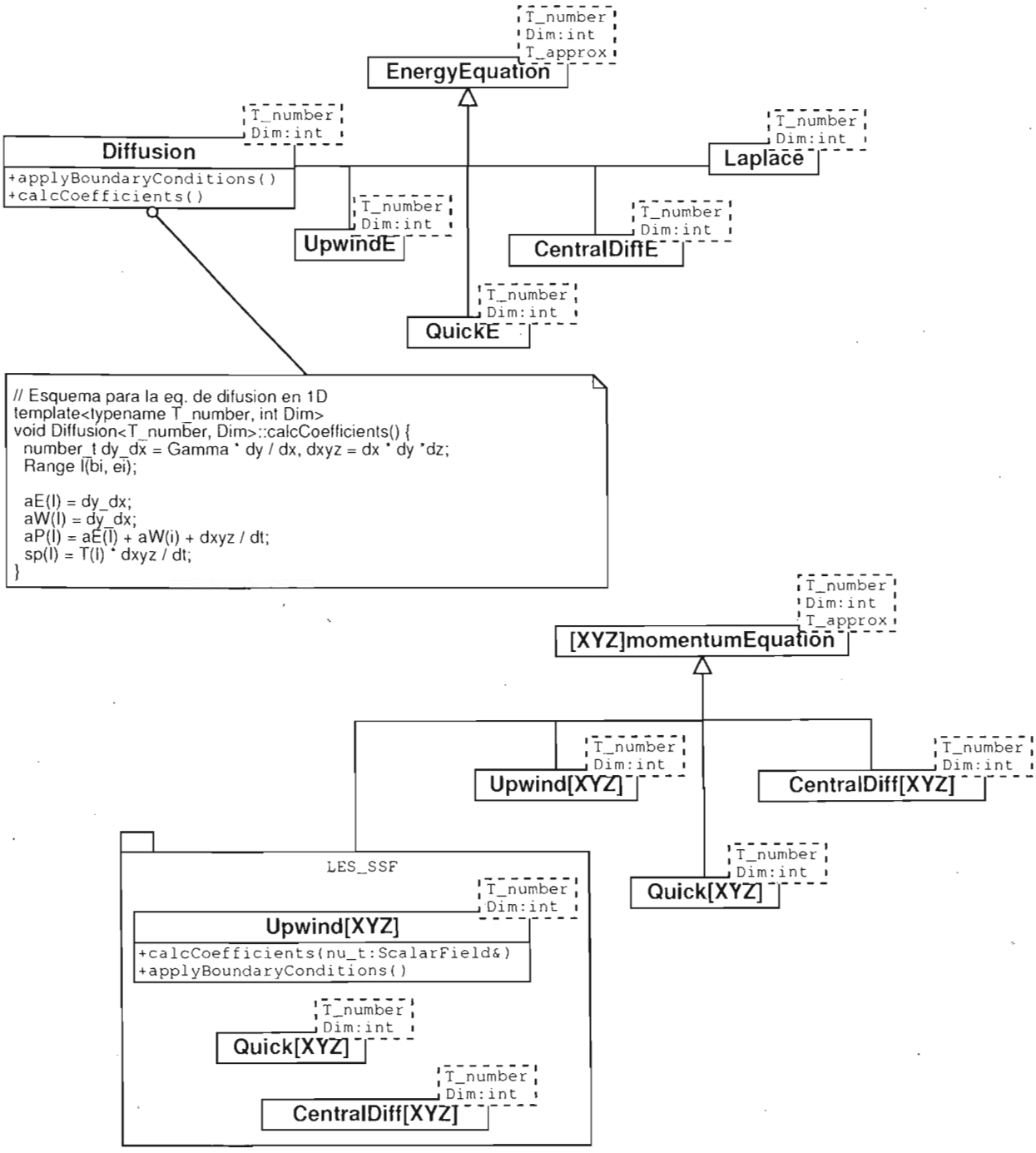


Figura A.5: Diagrama de clases: esquemas numéricos. Se puede observar que la implementación de un esquema numérico se realiza de manera similar a como se haría en Fortran 90.

Apéndice B

QUICK: mallas no uniformes

En esquema QUICK se utilizan polinomios de segundo grado para realizar la interpolación de los términos convectivos en las caras del volumen de control. Dependiendo del sentido de la velocidad en las caras del volumen de control, se determinan los puntos de la malla que participarán en la construcción de los polinomios, véase figura B.1.

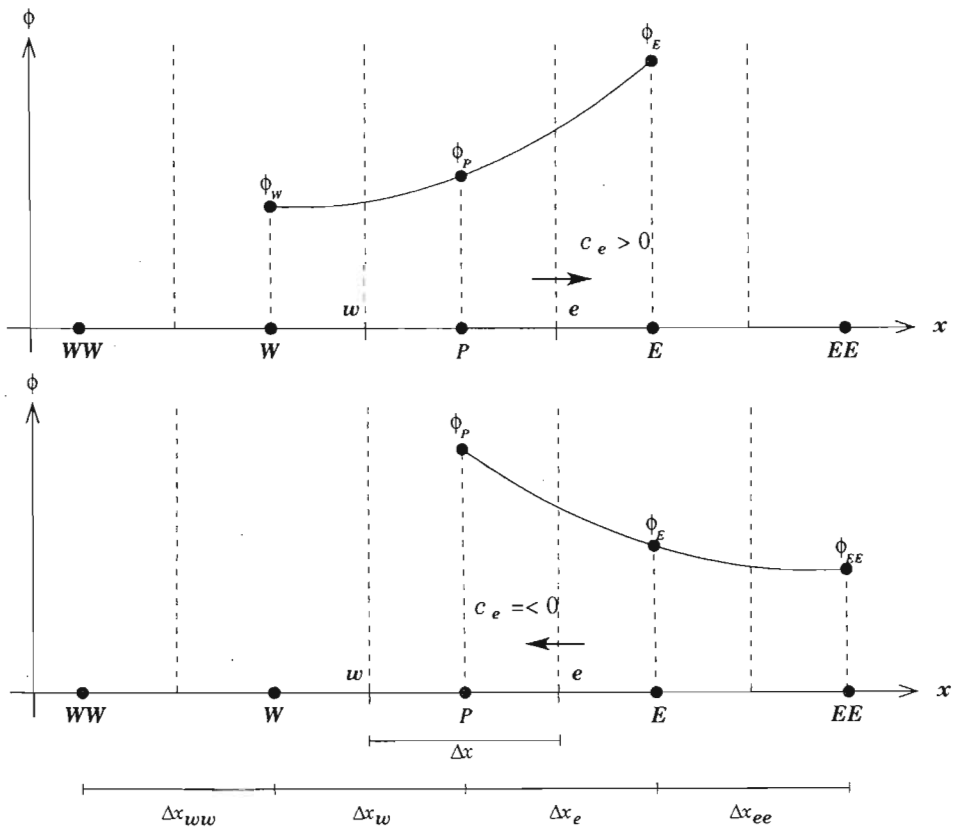


Figura B.1: Polinomios de segundo orden para el QUICK.

En todos los casos se debe encontrar un polinomio de la forma:

$$\phi(x) = ax^2 + bx + c \tag{B.1}$$

donde los coeficientes a, b y c deben ser calculados de acuerdo con los puntos que se usen para determinar cada polinomio.

Analizaremos primero el caso $\boxed{c_e > 0}$. En la figura B.1 se observa que los puntos que se deben usar para construir el polinomio son $\boldsymbol{W}$, $\boldsymbol{P}$ y $\boldsymbol{E}$, es decir, se utilizan los dos puntos que contienen a $\boldsymbol{e}$, $\boldsymbol{P}$ y $\boldsymbol{E}$, y un punto más de donde proviene el flujo (*upstream*), que en este caso es $\boldsymbol{W}$. Por lo tanto, las siguientes condiciones se deben cumplir:

$$\begin{aligned}
&\phi(x = 0) = \phi_{\boldsymbol{W}} = c,\\
&\phi(x = \Delta x_w) = \phi_{\boldsymbol{P}} = a\Delta x_w^2 + b\Delta x_w + c,\\
&\phi(x = \Delta x_w + \Delta x_e) = \phi_{\boldsymbol{E}} = a(\Delta x_w + \Delta x_e)^2 + b(\Delta x_w + \Delta x_e) + c.
\end{aligned}$$

Luego de un poco álgebra obtenemos el siguiente polinomio:

$$\phi_e = \frac{1}{2}\left(\phi_{\boldsymbol{P}} + \phi_{\boldsymbol{E}}\right) - \frac{1}{8}\left(\phi_{\boldsymbol{W}}\left(A_e^+ - B_e^+\right) - \phi_{\boldsymbol{P}} A_e^+ + \phi_{\boldsymbol{E}} B_e^+\right) \tag{B.2}$$

donde

$$A_e^+ = \frac{2\Delta x_e}{\Delta x_w}, \qquad \text{y} \qquad B_e^+ = -\frac{2\Delta x_e}{(\Delta x_w + \Delta x_e)},$$

En la ecuación (B.2) se observa que el polinomio contiene, en el primer término, una contribución que proviene de la interpolación lineal entre los dos puntos de la malla vecinos, $\boldsymbol{P}$ y $\boldsymbol{E}$.

Para el caso $\boxed{c_e \leq 0}$, se tiene que el polinomio se determina con los puntos $\boldsymbol{P}$, $\boldsymbol{E}$ y $\boldsymbol{EE}$, este último punto es de donde proviene el flujo, véase figura B.1. En este caso el polinomio que se obtiene es:

$$\phi_e = \frac{1}{2}\left(\phi_{\boldsymbol{P}} + \phi_{\boldsymbol{E}}\right) - \frac{1}{8}\left(\phi_{\boldsymbol{EE}}\left(A_e^- - B_e^-\right) - \phi_{\boldsymbol{E}} A_e^- + \phi_{\boldsymbol{P}} B_e^-\right) \tag{B.3}$$

donde

$$A_e^- = \frac{2\Delta x_e}{\Delta x_{ee}}, \qquad \text{y} \qquad B_e^- = -\frac{2\Delta x_e}{(\Delta x_e + \Delta x_{ee})},$$

Entonces, para mallas no uniformes el término convectivo $c_e\phi_e$ en el esquema QUICK se escribe como sigue:

$$c_e\phi_e = \frac{c_e}{2}(\phi_P + \phi_E) - c_{ep}\left(\phi_W\left(A_e^+ - B_e^+\right) - \phi_P A_e^+ + \phi_E B_e^+\right) \quad \text{(B.4)}$$

$$- c_{em}\left(\phi_{EE}\left(A_e^- - B_e^-\right) - \phi_E A_e^- + \phi_P B_e^-\right) \quad \text{(B.5)}$$

donde se han usado las funciones definidas en la ecuación (4.18) y en la figura 4.5.

Siguiendo el mismo proceso se obtienen los polinomios correspondientes para los términos $c_w\phi_w$, $c_n\phi_n$, $c_s\phi_s$, $c_f\phi_f$ y $c_b\phi_b$. Por ejemplo, el polinomio para $c_w\phi_w$ se escribe como sigue:

$$c_w\phi_w = \frac{c_w}{2}(\phi_P + \phi_W) - c_{wp}\left(\phi_{WW}\left(A_w^+ - B_w^+\right) - \phi_W A_w^+ + \phi_P B_w^+\right) \quad \text{(B.6)}$$

$$- c_{wm}\left(\phi_E\left(A_w^- - B_w^-\right) - \phi_P A_w^- + \phi_W B_w^-\right) \quad \text{(B.7)}$$

donde los coeficientes se definen como sigue:

$$A_w^+ = \frac{2\Delta x_w}{\Delta x_{ww}}, \quad B_w^+ = -\frac{2\Delta x_w}{(\Delta x_{ww} + \Delta x_w)}, \quad A_w^- = \frac{2\Delta x_w}{\Delta x_e}, \quad B_w^- = -\frac{2\Delta x_w}{(\Delta x_e + \Delta x_w)},$$

El término $c_e\phi_e - c_w\phi_w$ que aparece en la ecuación (4.2) con las definiciones anteriores se escribe:

$$\begin{aligned} c_e\phi_e - c_w\phi_w &= \phi_P\left(\frac{c_e}{2} + c_{ep}A_e^+ - c_{em}B_e^- - \frac{c_w}{2} + c_{wp}B_w^+ - c_{wm}A_w^-\right) \quad \text{(B.8)} \\ &- \phi_E\left(-\frac{c_e}{2} + c_{ep}B_e^+ - c_{em}A_e^- - c_{wm}\left(A_w^- - B_w^-\right)\right) \\ &- \phi_W\left(-\frac{c_w}{2} + c_{wp}A_w^+ - c_{wm}B_w^- + c_{ep}\left(A_e^+ - B_e^+\right)\right) \\ &- \phi_{EE}c_{em}\left(A_e^- - B_e^-\right) - \phi_{WW}c_{wp}\left(A_w^+ - B_w^+\right) \end{aligned}$$

Por lo tanto, los coeficientes convectivos en esta formulación son de la forma:

$$C_E = -\frac{c_e}{2} + c_{ep}B_e^+ - c_{em}A_e^- - c_{wm}\left(A_w^- - B_w^-\right) \quad \text{(B.9)}$$

$$C_W = -\frac{c_w}{2} + c_{wp}A_w^+ - c_{wm}B_w^- + c_{ep}\left(A_e^+ - B_e^+\right) \quad \text{(B.10)}$$

Haciendo un tratamiento similar tenemos que

$$C_N = -\frac{c_n}{2} + c_{np}B_n^+ - c_{nm}A_n^- - c_{sm}\left(A_s^- - B_s^-\right) \quad \text{(B.11)}$$

$$C_S = -\frac{c_s}{2} + c_{sp}A_s^+ - c_{sm}B_s^- + c_{np}\left(A_n^+ - B_n^+\right) \quad \text{(B.12)}$$

$$C_F = -\frac{c_f}{2} + c_{fp}B_f^+ - c_{fm}A_f^- - c_{bm}\left(A_b^- - B_b^-\right) \quad \text{(B.13)}$$

$$C_B = -\frac{c_b}{2} + c_{bp}A_b^+ - c_{bm}B_b^- + c_{fp}\left(A_f^+ - B_f^+\right) \quad \text{(B.14)}$$

y

$$\begin{aligned} C_P = C_E &+ C_W + C_N + C_S + C_F + C_B \\ &+ (c_e - c_w) + (c_n - c_s) + (c_f - c_b) + \\ &+ c_{em}\left(A_e^- - B_e^-\right) - c_{wp}\left(A_w^+ - B_w^+\right) \\ &+ c_{nm}\left(A_n^- - B_n^-\right) - c_{sp}\left(A_s^+ - B_s^+\right) \\ &+ c_{fm}\left(A_f^- - B_f^-\right) - c_{bp}\left(A_b^+ - B_b^+\right) \end{aligned} \tag{B.15}$$

Cuando las mallas son uniformes los coeficientes A_{nb}^{+-} son iguales a 2, mientras que los términos B_{nb}^{+-} son iguales a 1. En este caso se recuperan los coeficientes descritos en la sección 4.3.3.